CS 225

Data Structures

April 12 – Graphs

G Carl Evans

In Review: Data Structures

Array
- Sorted Array
- Unsorted Array
  - Stacks
  - Queues
  - Hashing
  - Heaps
    - Priority Queues
  - UpTrees
    - Disjoint Sets

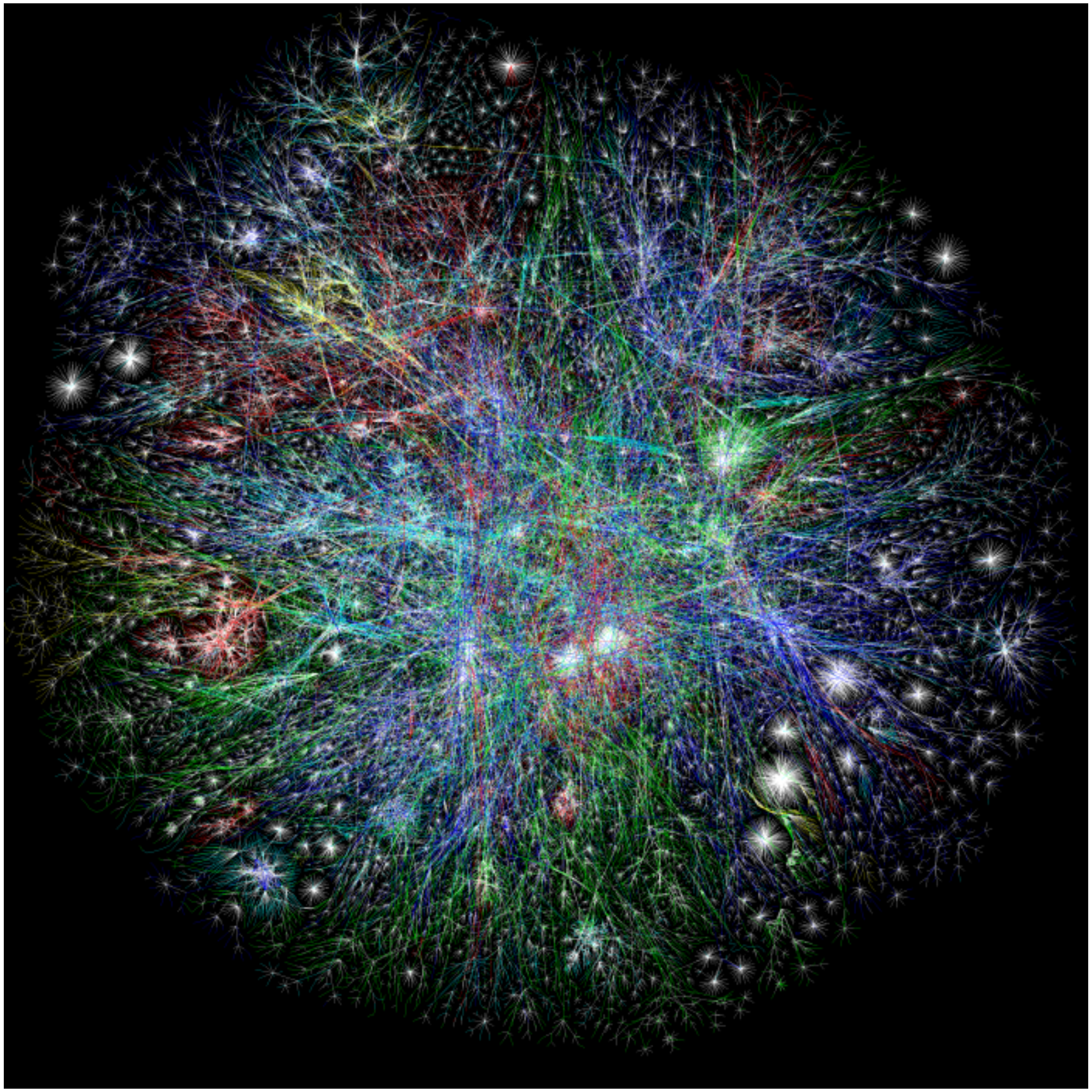
Linked
- Doubly Linked List
- Trees
  - BTree
  - Binary Tree
    - Huffman Encoding
    - kd-Tree
    - AVL Tree

In Review: Data Structures

Array
- Sorted Array
- Unsorted Array
  - Stacks
  - Queues
  - Hashing
  - Heaps
    - Priority Queues
  - UpTrees
    - Disjoint Sets

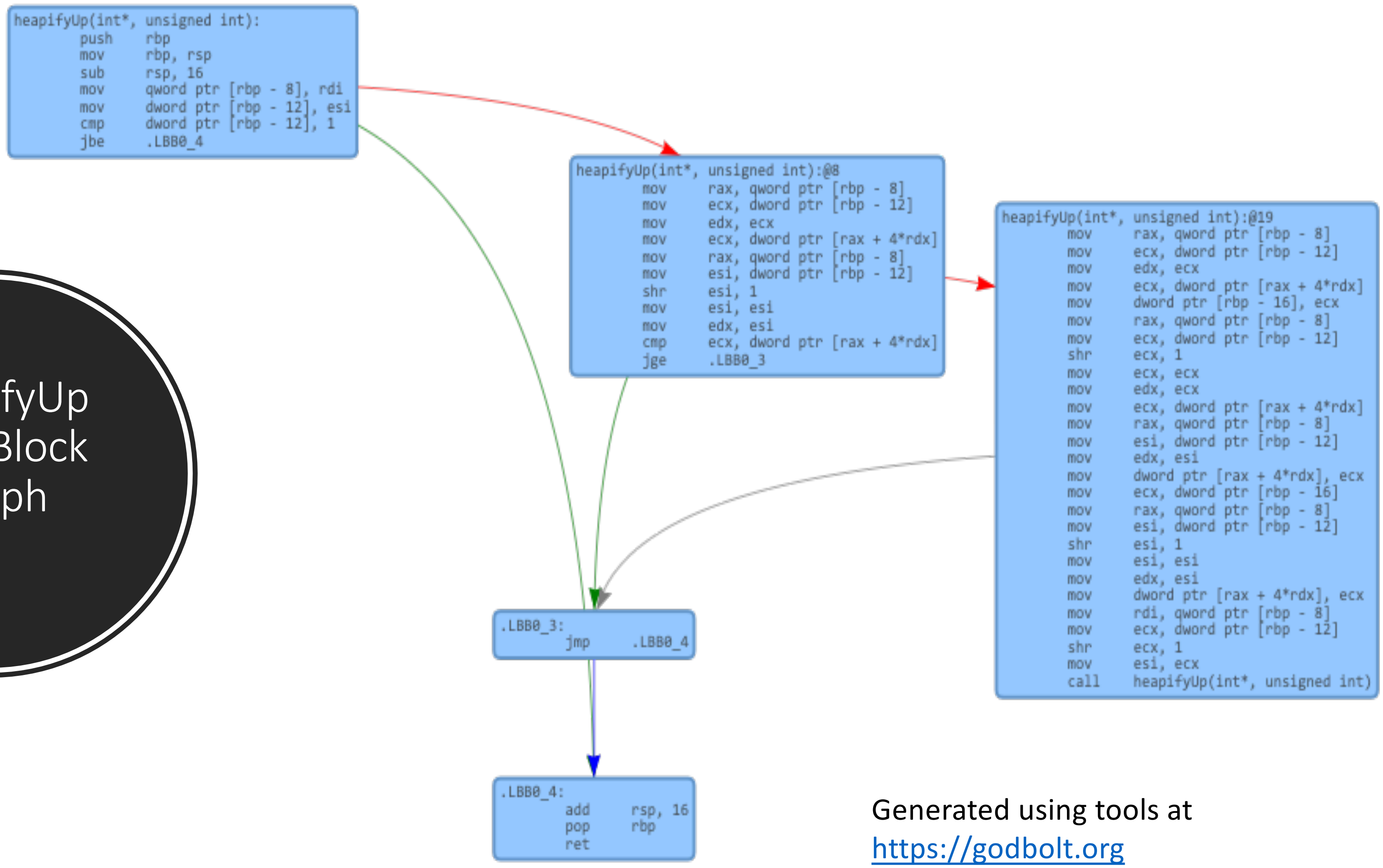
Graphs

Linked
- Doubly Linked List
- Skip List
- Trees
  - BTree
  - Binary Tree
    - Huffman Encoding
    - kd-Tree
    - AVL Tree

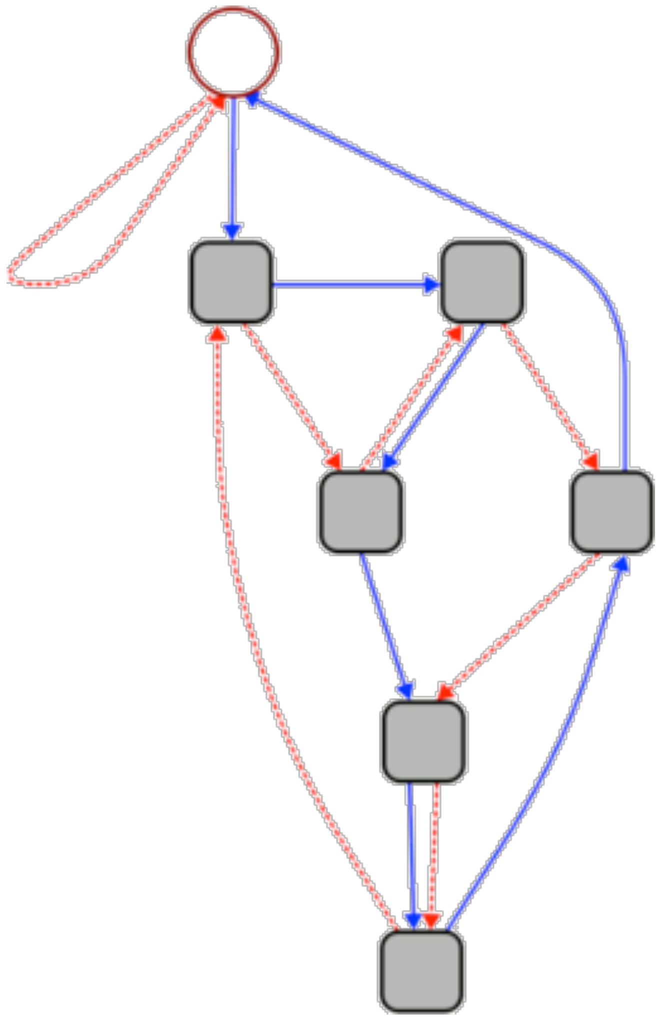


The Internet 2003
The OPTE Project (2003)
Map of the entire internet; nodes are routers; edges are connections.

HeapifyUp BasicBlock Graph



Generated using tools at https://godbolt.org



This graph can be used to quickly calculate whether a given number is divisible by 7.

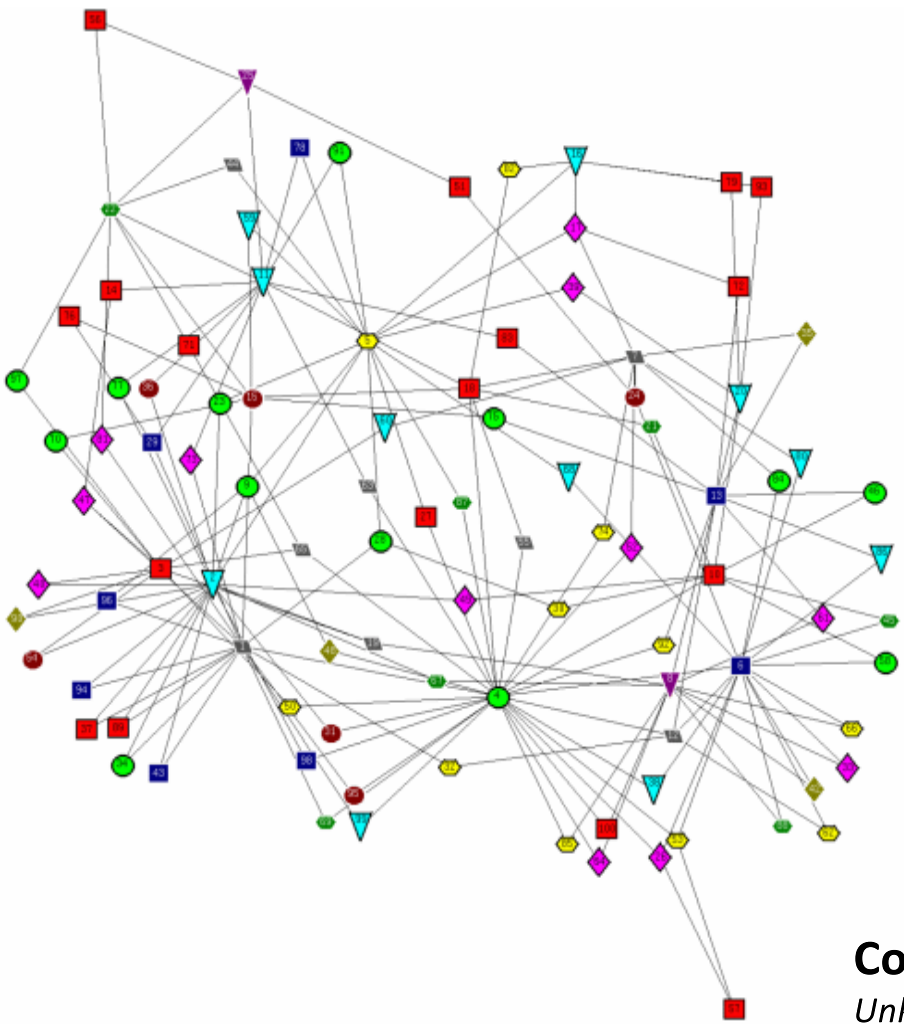
1. Start at the circle node at the top.
2. For each digit **d** in the given number, follow **d** blue (solid) edges in succession. As you move from one digit to the next, follow **1** red (dashed) edge.
3. If you end up back at the circle node, your number is divisible by 7.

3703

"Rule of 7"

Unknown Source

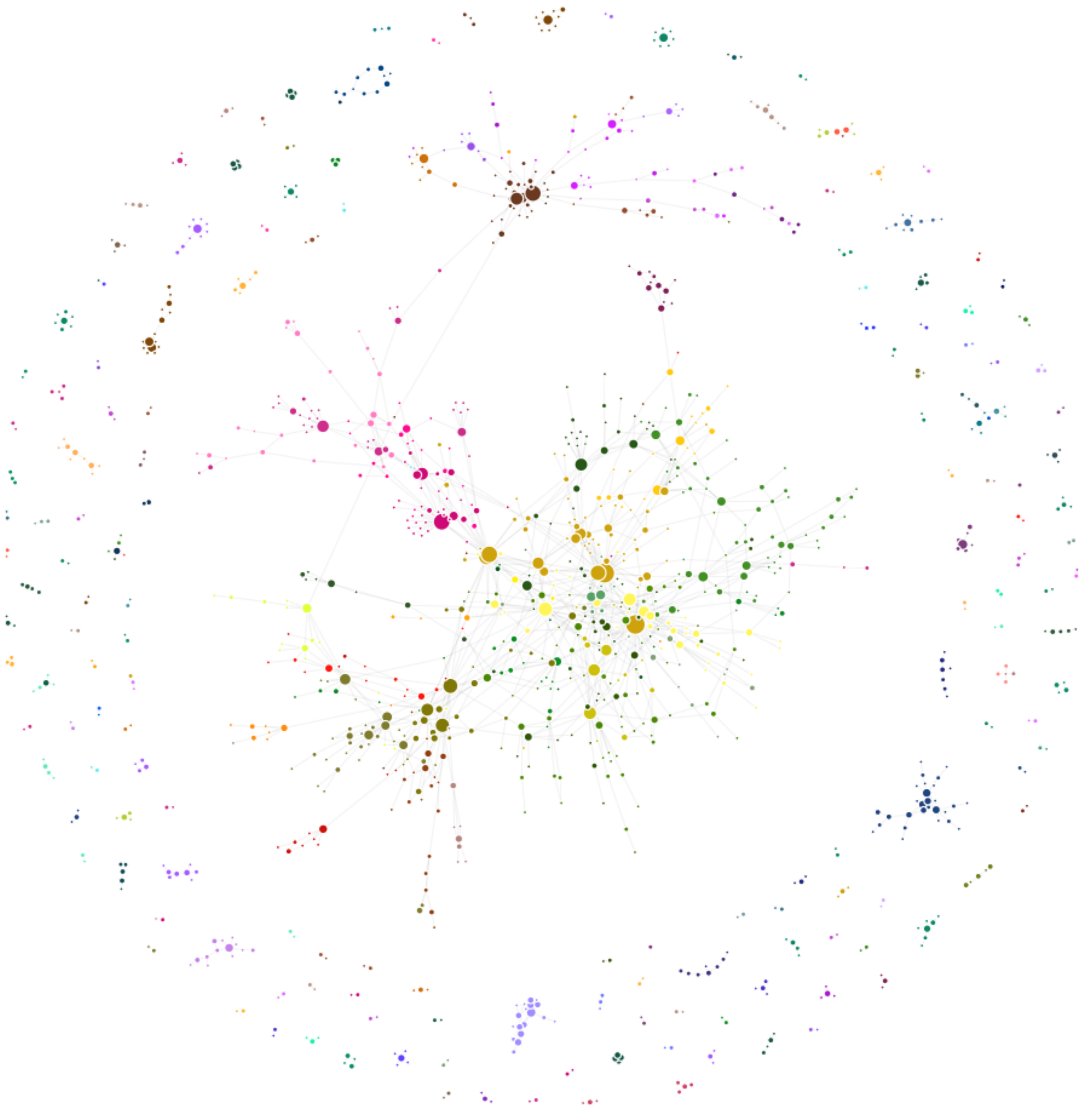
Presented by Cinda Heeren, 2016



Conflict-Free Final Exam Scheduling Graph

Unknown Source

Presented by Cinda Heeren, 2016

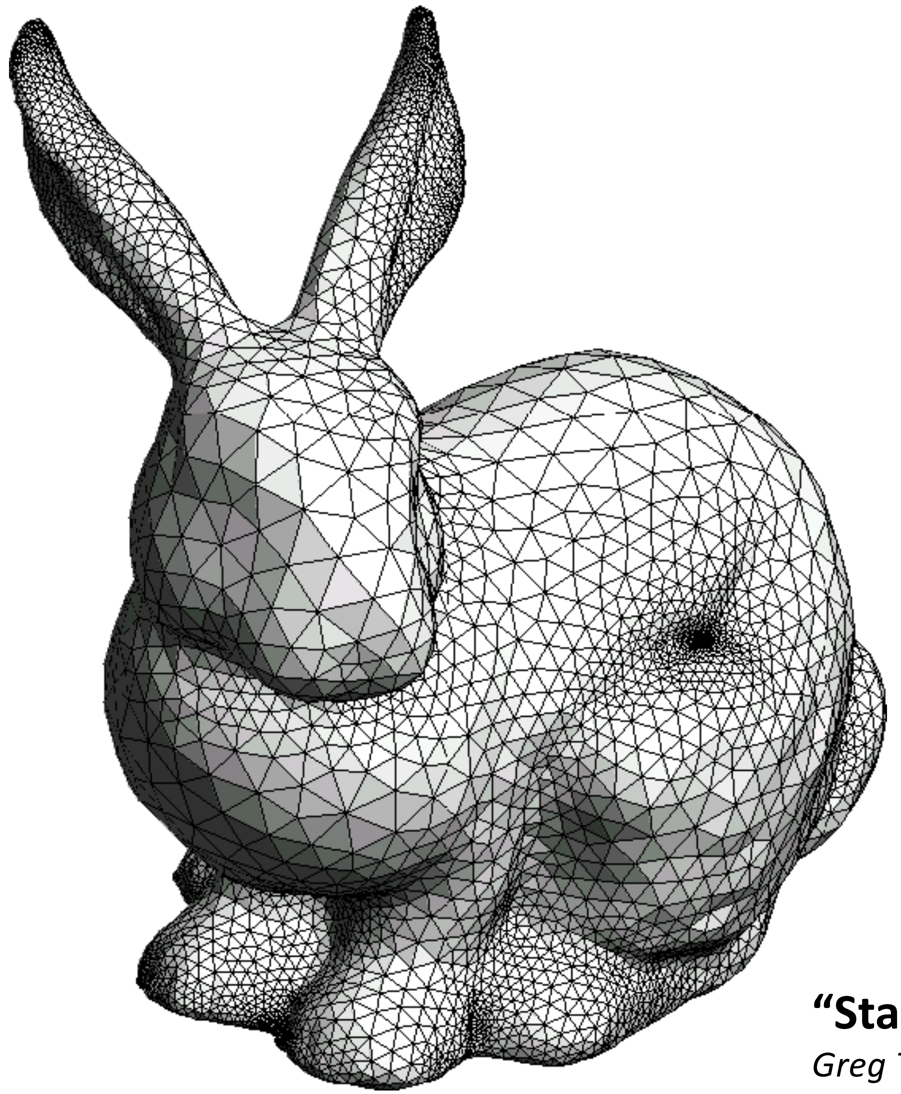


Class Hierarchy At University of Illinois Urbana-Champaign

A. Mori, W. Fagen-Ulmschneider, C. Heeren

Graph of every course at UIUC; nodes are courses, edges are prerequisites

http://waf.cs.illinois.edu/discovery/class_hierarchy_at_illinois/

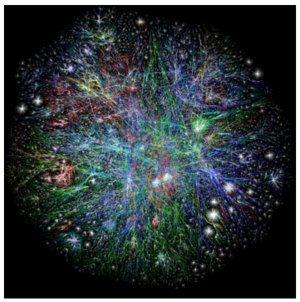


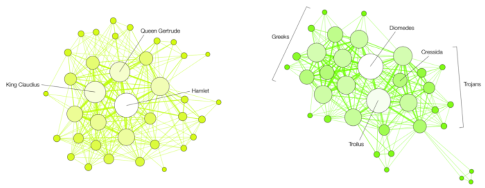
“Stanford Bunny”
Greg Turk and Mark Levoy (1994)

HAMLET

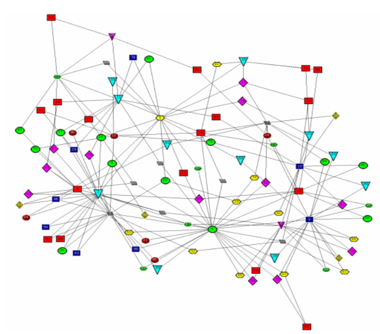
TROILUS AND CRESSIDA

Graphs



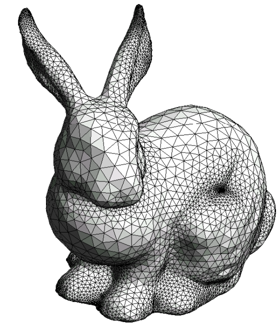


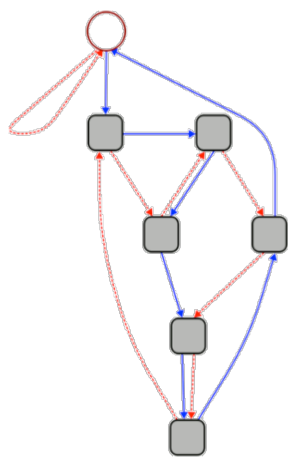
HAMLET TROILUS AND CRESSIDA

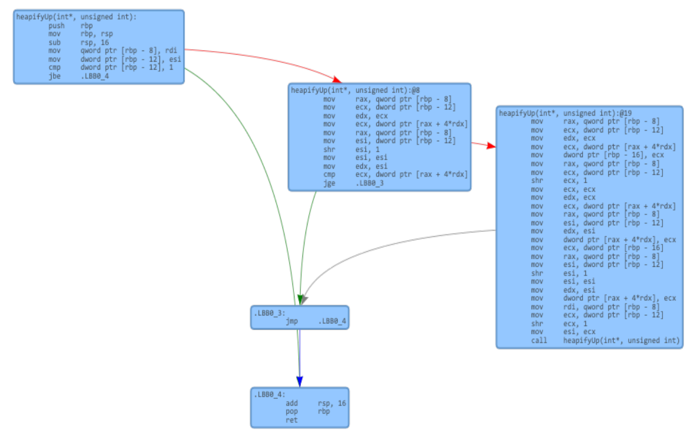


To study all of these structures:

1. A common vocabulary
2. Graph implementations
3. Graph traversals
4. Graph algorithms

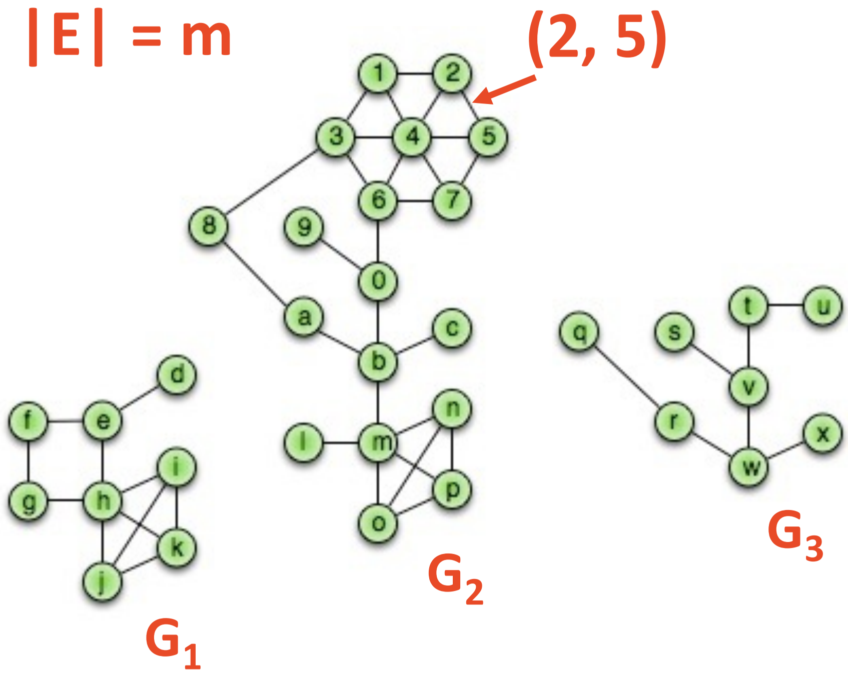






Graph Vocabulary

$G = (V, E)$
$|V| = n$
$|E| = m$



Incident Edges:
$I(v) = \{ \{x, v\} \text{ in } E \}$

Degree(v): $|I|$

Adjacent Vertices:
$A(v) = \{ x : \{x, v\} \text{ in } E \}$

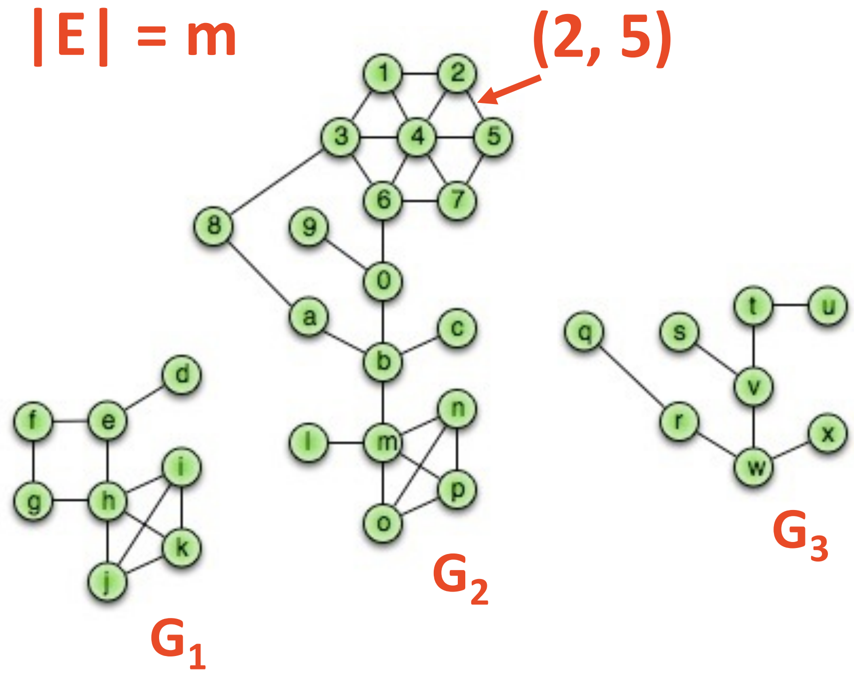
Path(G_2): Sequence of vertices connected by edges

Cycle(G_1): Path with a common begin and end vertex.

Simple Graph(G): A graph with no self loops or multi-edges.

Graph Vocabulary

G = (V, E)
|V| = n
|E| = m



Subgraph(G):
G' = (V', E'):
V' ∈ V, E' ∈ E, and
(u, v) ∈ E' → u ∈ V', v ∈ V'

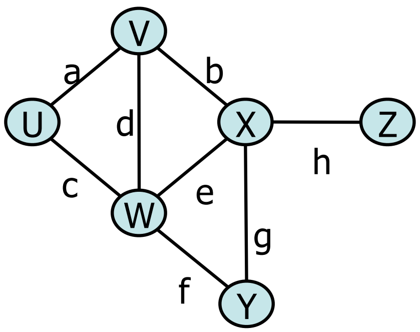
Complete subgraph(G)
Connected subgraph(G)
Connected component(G)
Acyclic subgraph(G)
Spanning tree(G)

Running times are often reported by **n**, the number of vertices, but often depend on **m**, the number of edges.

How many edges? **Minimum edges:**

Not Connected:

Connected*:



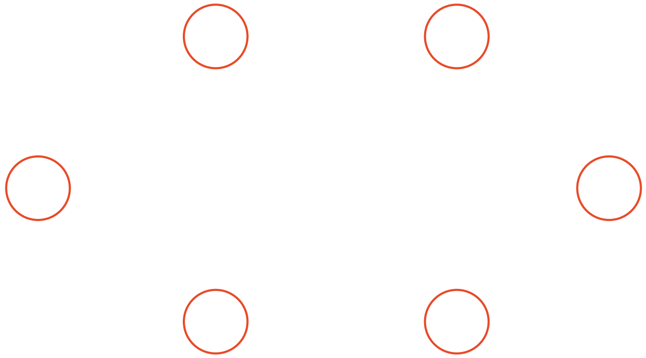
Maximum edges:

Simple:

Not simple:

$$\sum_{v \in V} \deg(v) =$$

Connected Graphs



Proving the size of a minimally connected graph

Theorem:
Every connected graph **G=(V, E)** has at least **|V|-1** edges.

Thm: Every connected graph **G=(V, E)** has at least **|V|-1** edges.

Proof: Consider an arbitrary, connected graph **G=(V, E)**.

Suppose |V| = 1:

Definition: A connected graph of 1 vertex has 0 edges.

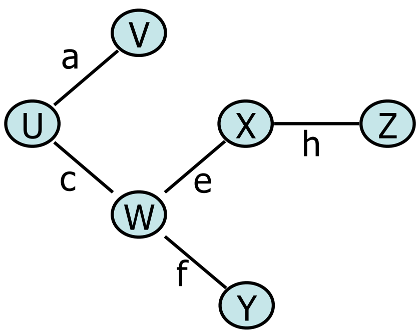
Theorem: |V|-1 edges ➔ 1-1 = 0.

Inductive Hypothesis: For any $\mathbf{j < |V|}$, any connected graph of $\mathbf{j}$ vertices has at least $\mathbf{j-1}$ edges.

Suppose |V| > 1:

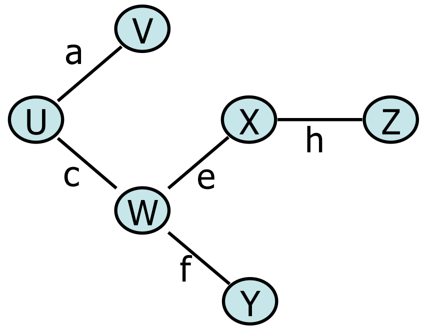
1. Choose any vertex:

2. Partition:



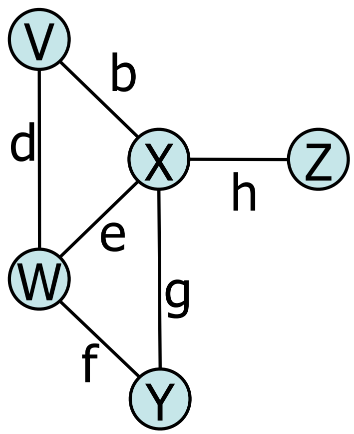
Suppose |V| > 1:

3. Count the edges



Graph ADT

Data:
- Vertices
- Edges
- Some data structure maintaining the structure between vertices and edges.

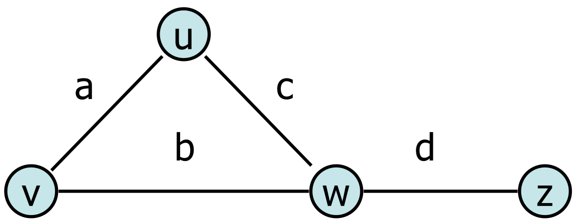


Functions:
- insertVertex(K key);
- insertEdge(Vertex v1, Vertex v2, K key);

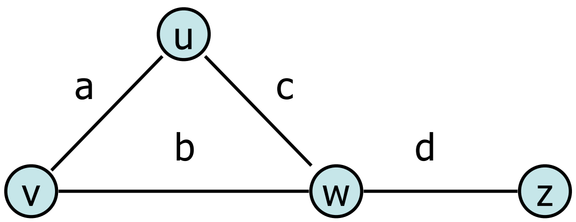
- removeVertex(Vertex v);
- removeEdge(Vertex v1, Vertex v2);

- incidentEdges(Vertex v);
- areAdjacent(Vertex v1, Vertex v2);

Graph Implementation Idea



Graph Implementation: Edge List



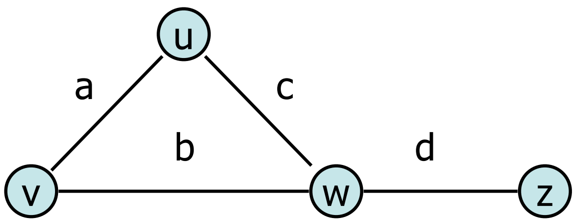
Vertex Collection:

u
v
w
z

Edge Collection:

u	v	a
v	w	b
u	w	c
w	z	d

Graph Implementation: Edge List



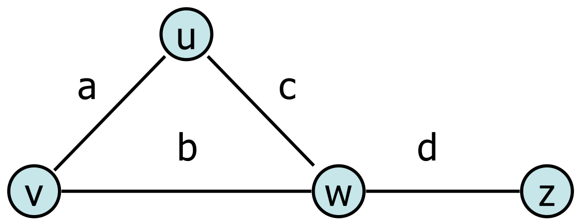
u
v
w
z

u	v	a
v	w	b
u	w	c
w	z	d

insertVertex(K key):

removeVertex(Vertex v):

Graph Implementation: Edge List



u
v
w
z

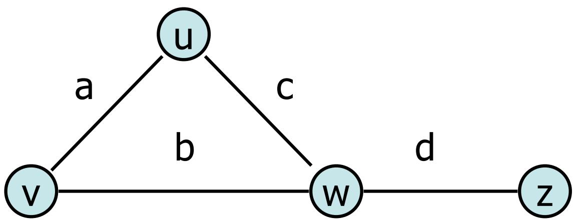
u	v	a
v	w	b
u	w	c
w	z	d

incidentEdges(Vertex v):

areAdjacent(Vertex v1, Vertex v2):

```
G.incidentEdges(v1).contains(v2)
```

Graph Implementation: Edge List

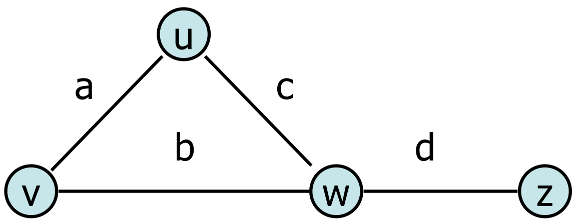


u
v
w
z

u	v	a
v	w	b
u	w	c
w	z	d

insertEdge(Vertex v1, Vertex v2, K key):

Graph Implementation: Adjacency Matrix

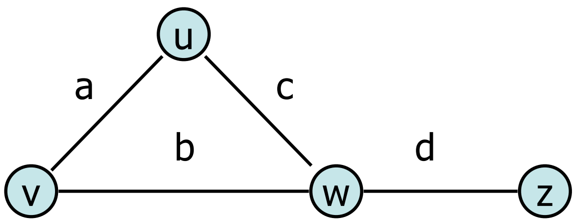


u
v
w
z

u	v	a
v	w	b
u	w	c
w	z	d

	u	v	w	z
u				
v				
w				
z				

Graph Implementation: Adjacency Matrix

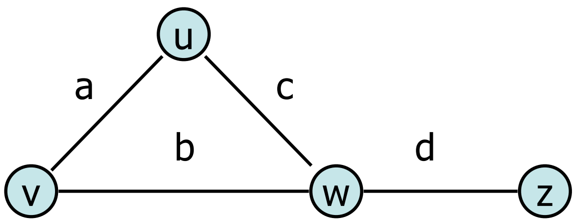


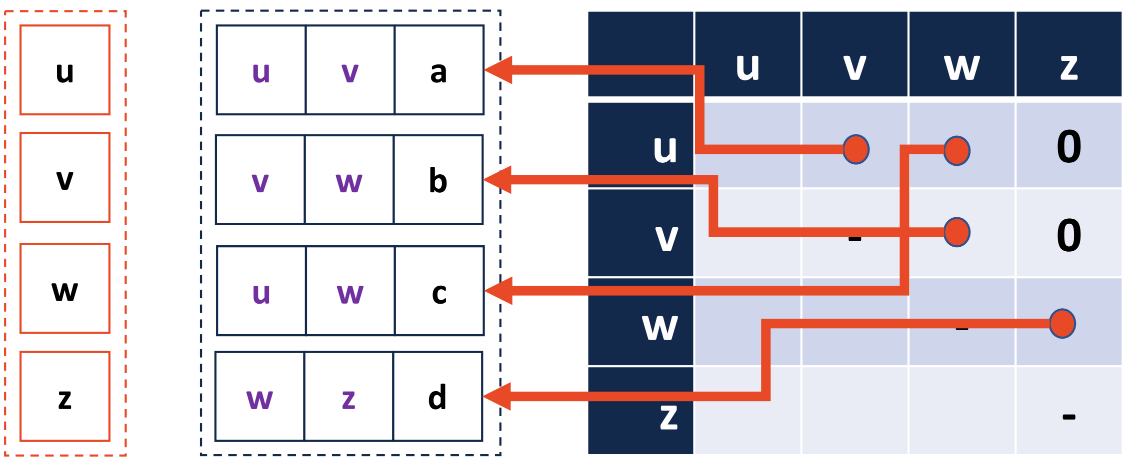
u
v
w
z

u	v	a
v	w	b
u	w	c
w	z	d

	u	v	w	z
u	-	1	1	0
v		-	1	0
w			-	1
z				-

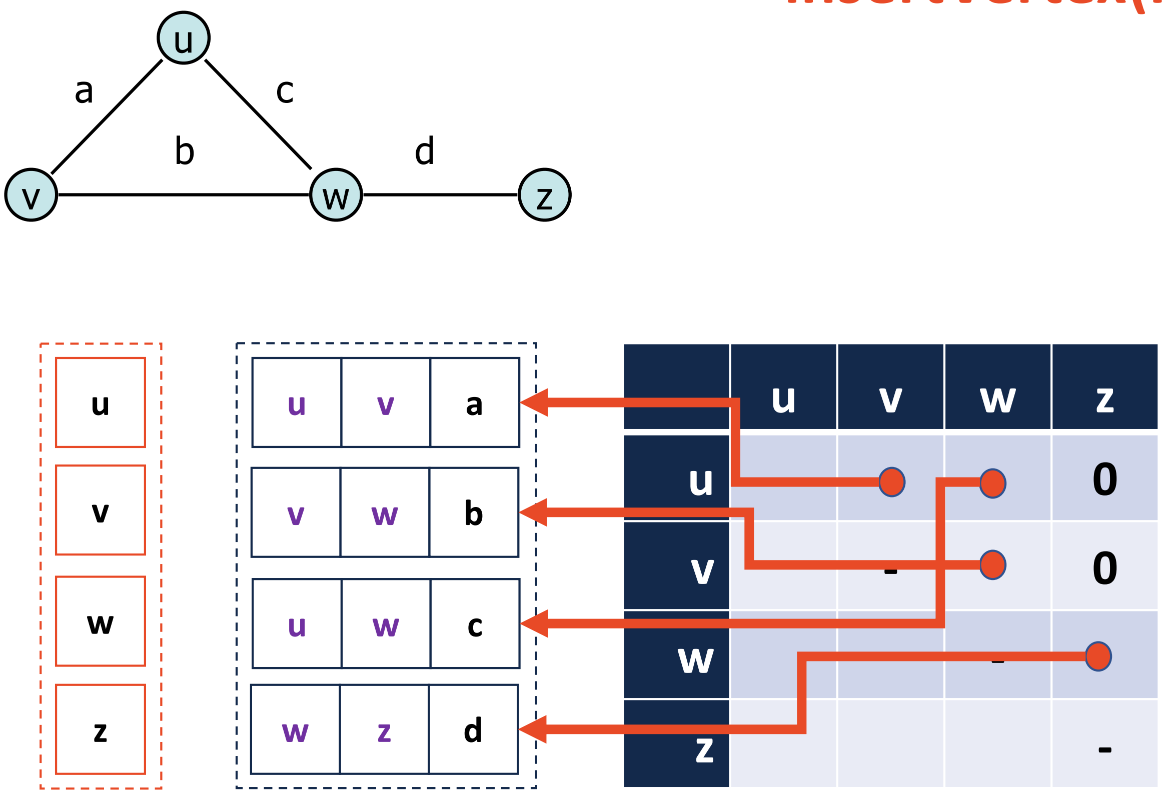
Graph Implementation: Adjacency Matrix





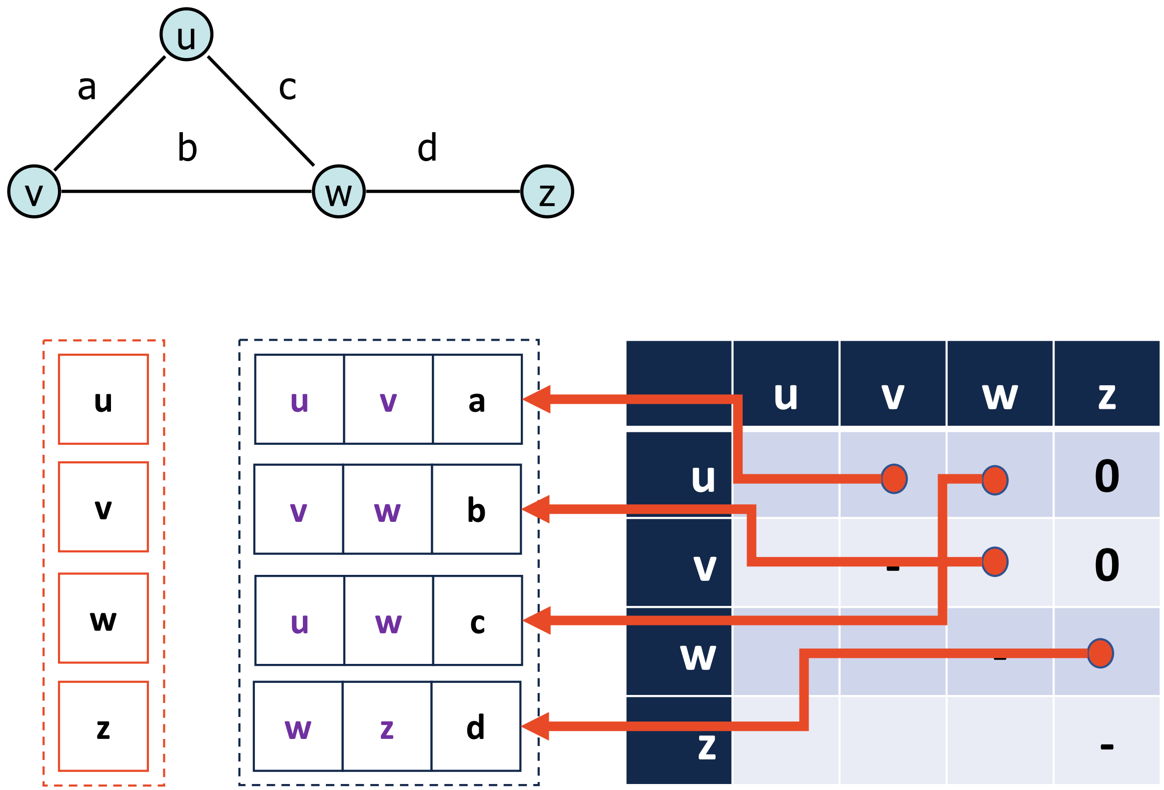
Graph Implementation: Adjacency Matrix

insertVertex(K key):



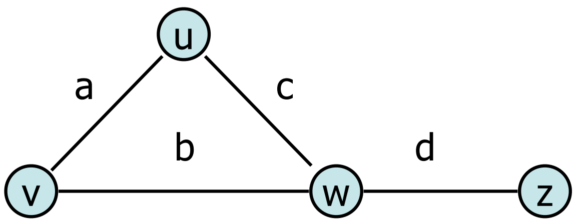
Graph Implementation: Adjacency Matrix

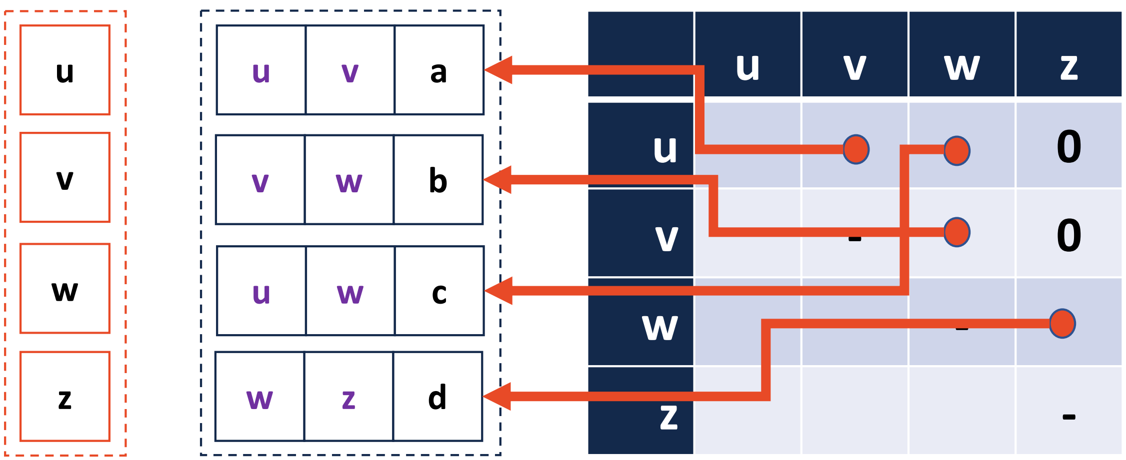
removeVertex(Vertex v):



Graph Implementation: Adjacency Matrix

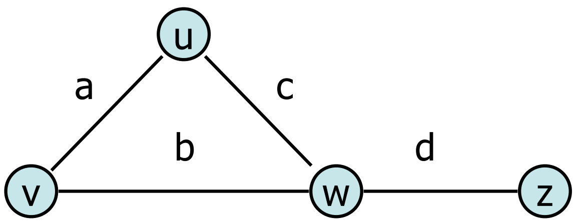
incidentEdges(Vertex v):

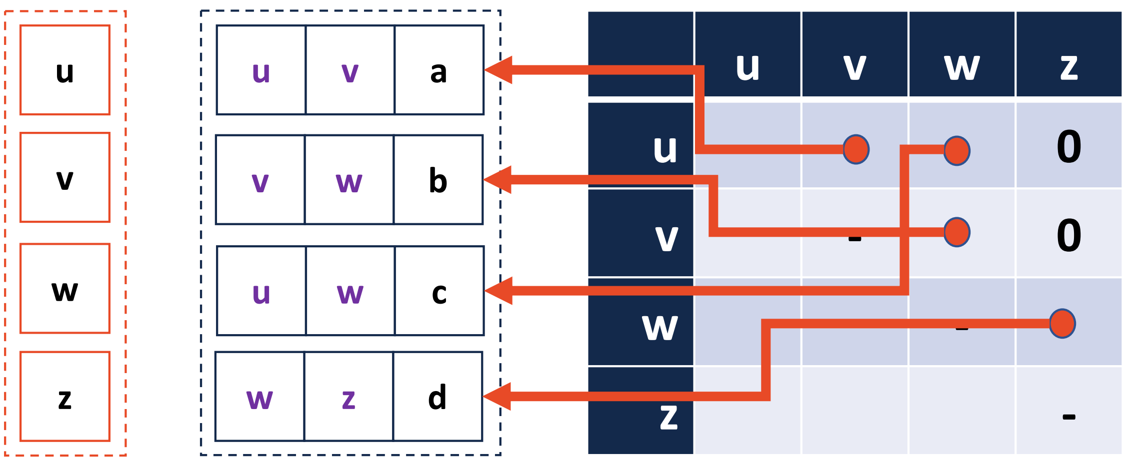




Graph Implementation: Adjacency Matrix

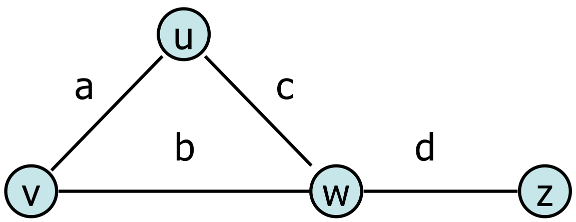
areAdjacent(Vertex v1, Vertex v2):

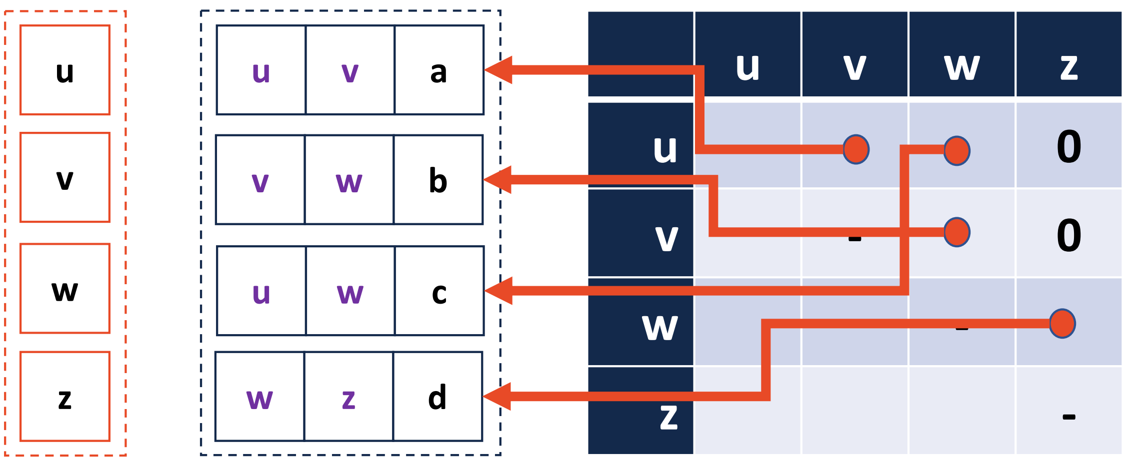




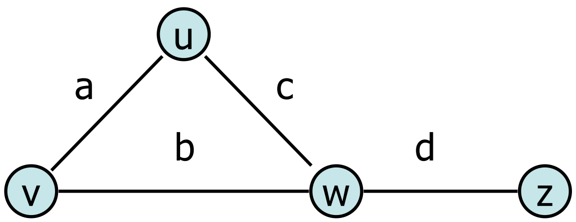
Graph Implementation: Adjacency Matrix

insertEdge(Vertex v1, Vertex v2, K key):





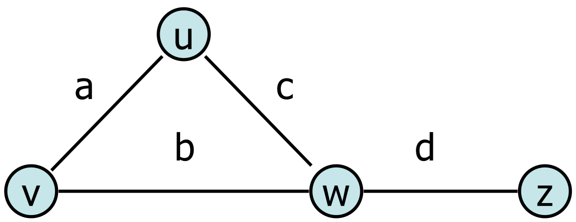
Graph Implementation: Edge List

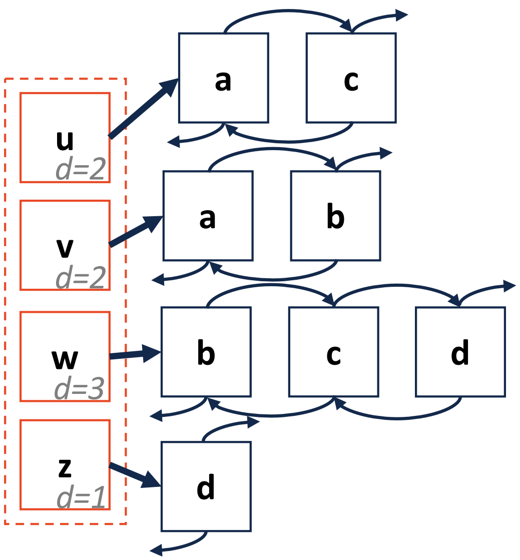


u
v
w
z

u	v	a
v	w	b
u	w	c
w	z	d

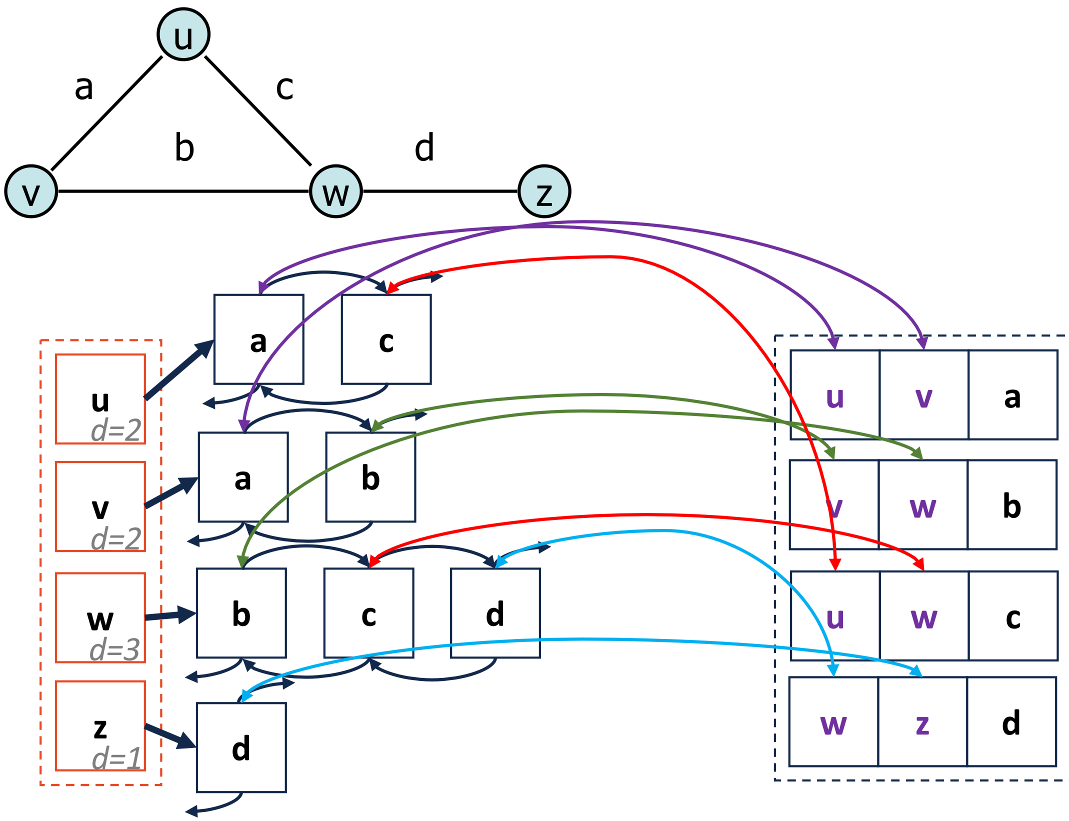
Adjacency List





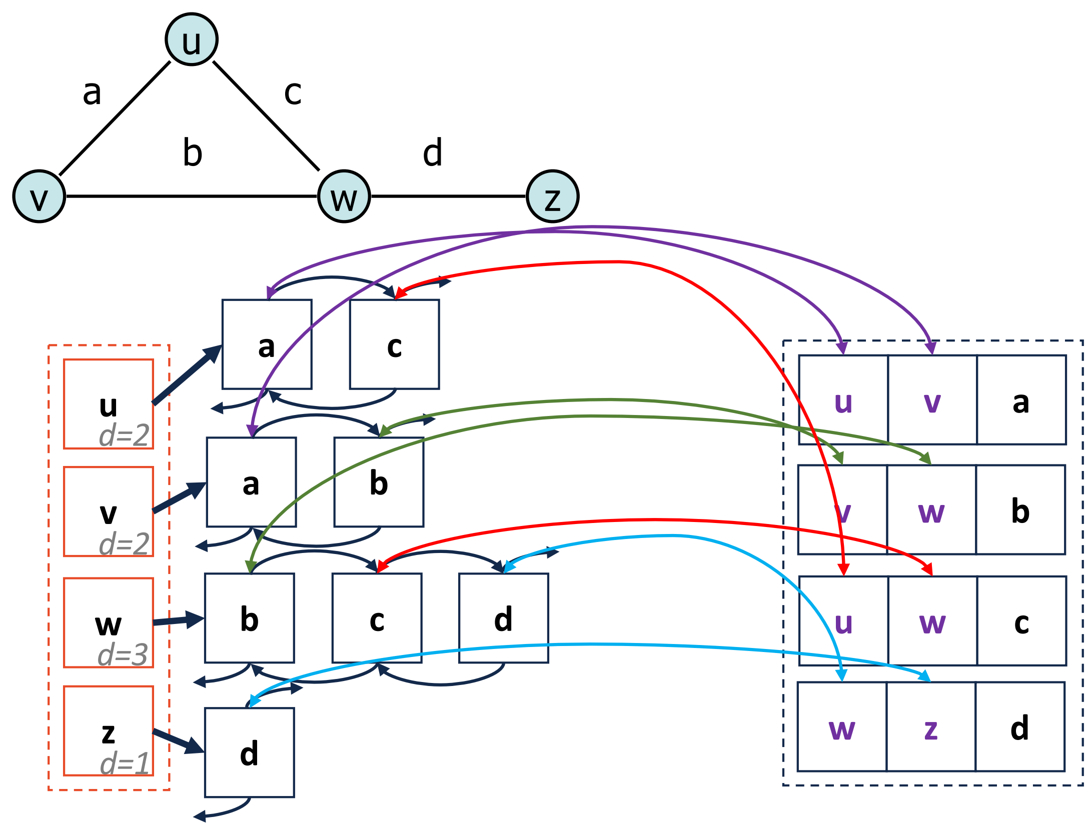
u	v	a
v	w	b
u	w	c
w	z	d

Adjacency List



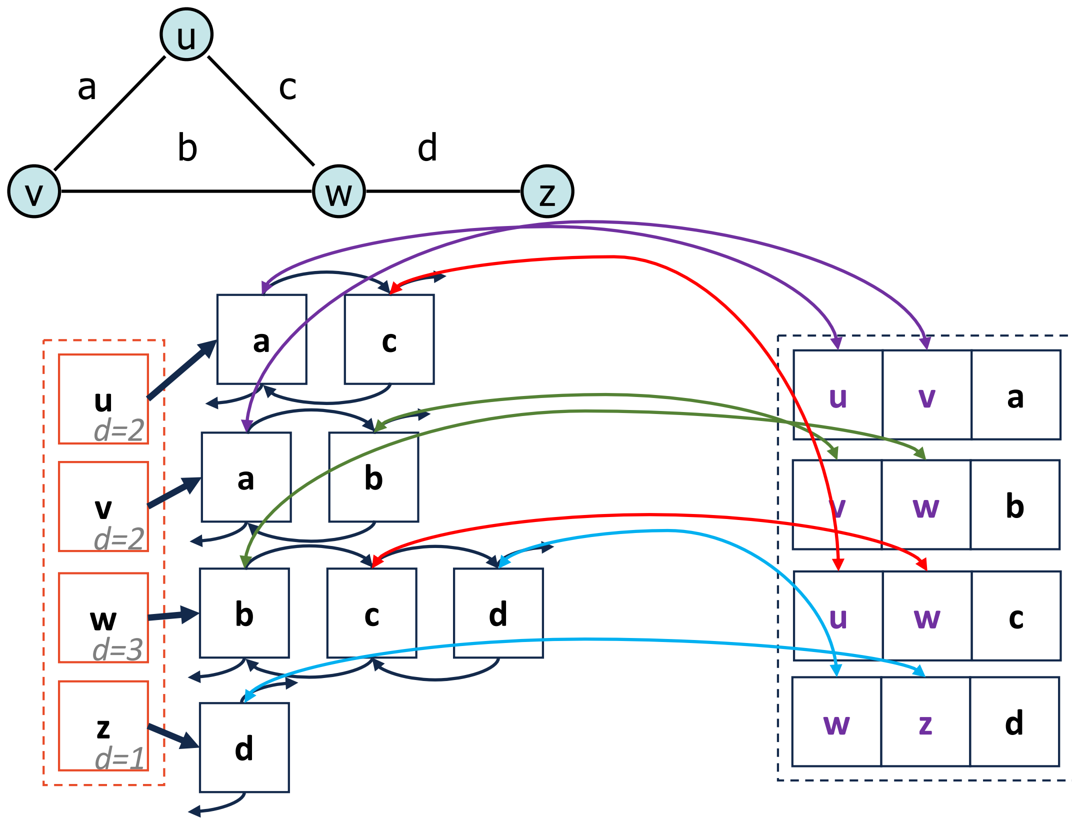
Adjacency List

insertVertex(K key):



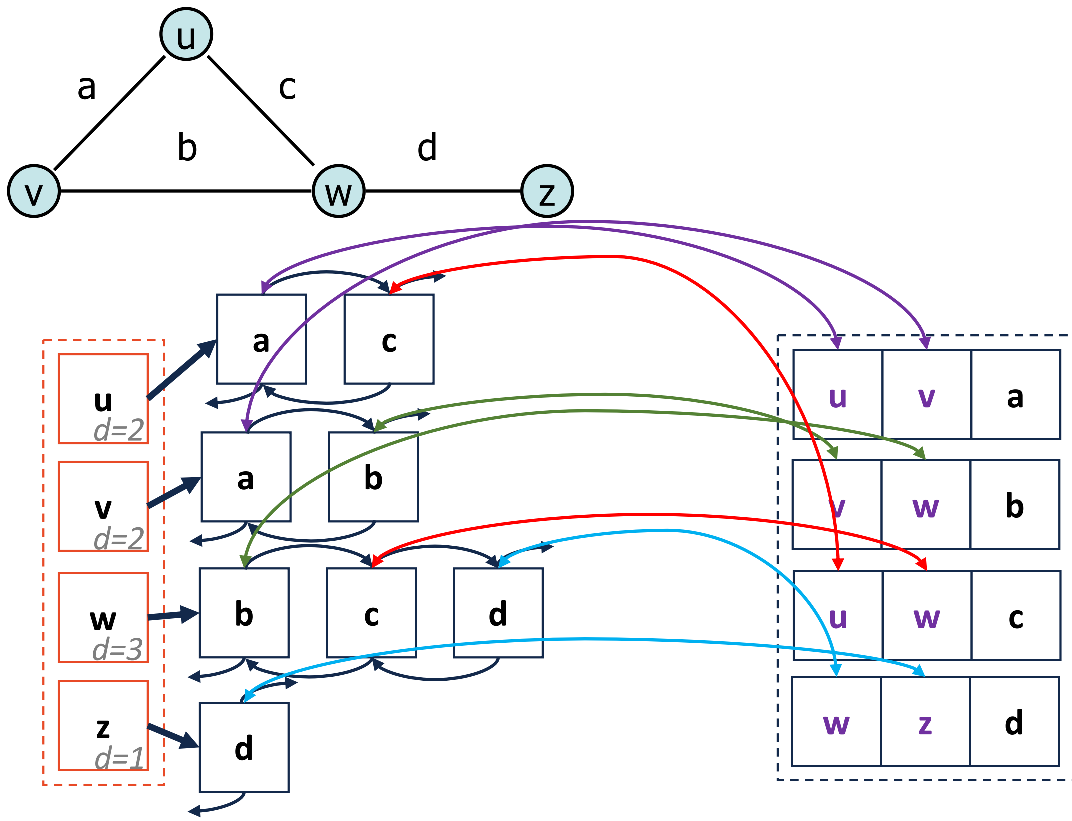
Adjacency List

removeVertex(Vertex v):



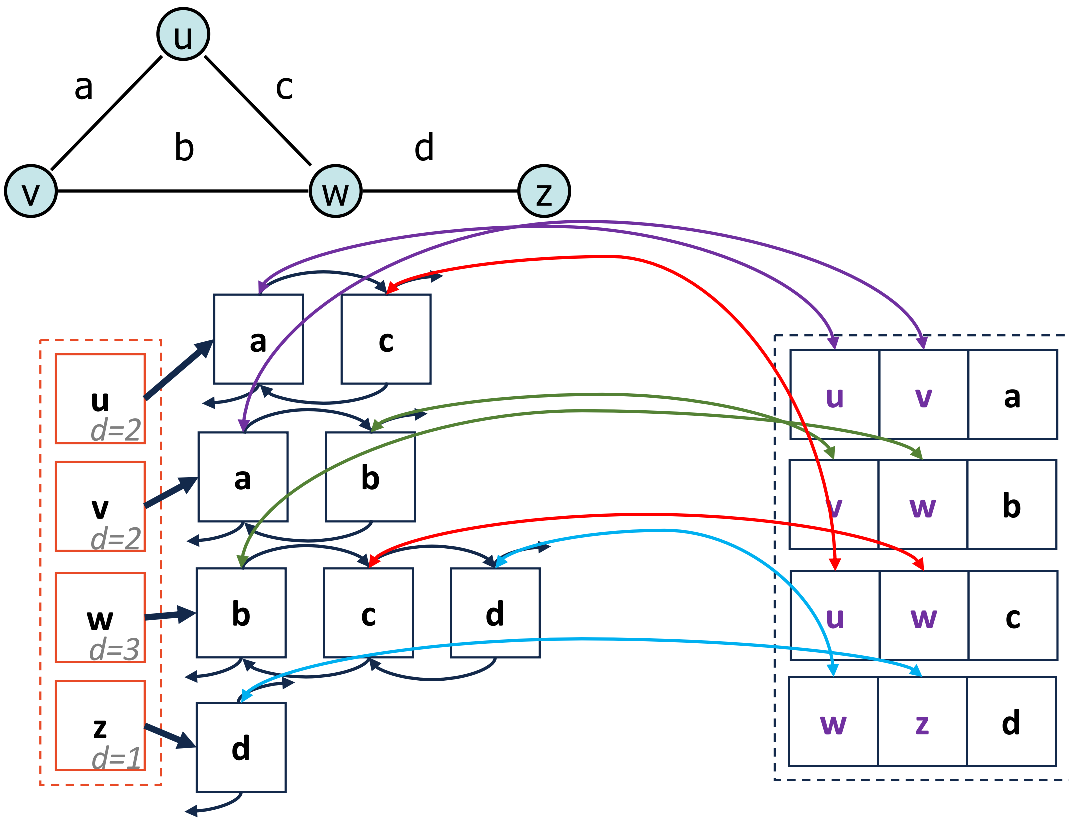
Adjacency List

incidentEdges(Vertex v):



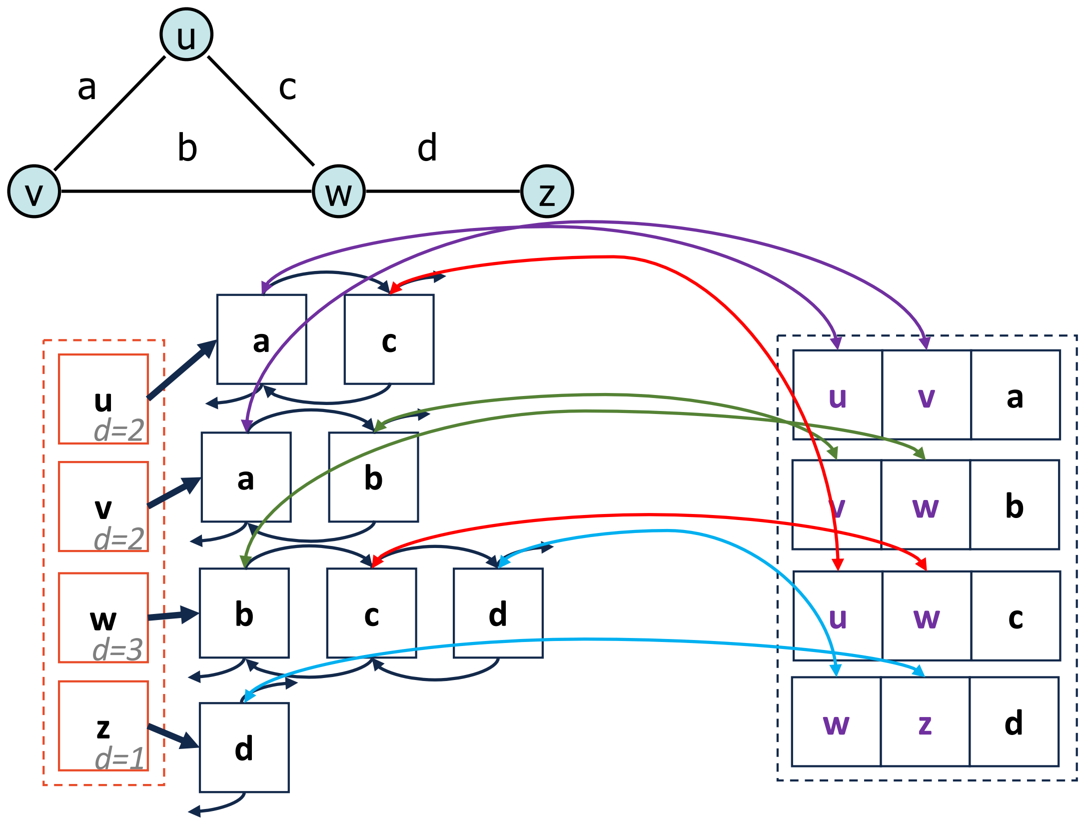
Adjacency List

areAdjacent(Vertex v1, Vertex v2):



Adjacency List

insertEdge(Vertex v1, Vertex v2, K key):



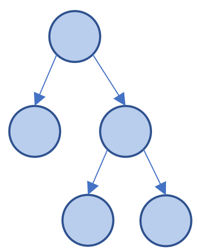
Expressed as O(f)	Edge List	Adjacency Matrix	Adjacency List
Space	n+m	n^2	n+m
insertVertex(v)	1	n	1
removeVertex(v)	m	n	deg(v)
insertEdge(v, w, k)	1	1	1
removeEdge(v, w)	1	1	1
incidentEdges(v)	m	n	deg(v)
areAdjacent(v, w)	m	1	min(deg(v), deg(w))

Traversal:

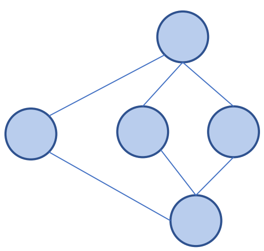
Objective: Visit every vertex and every edge in the graph.

Purpose: Search for interesting sub-structures in the graph.

We've seen traversal before ….but it's different:

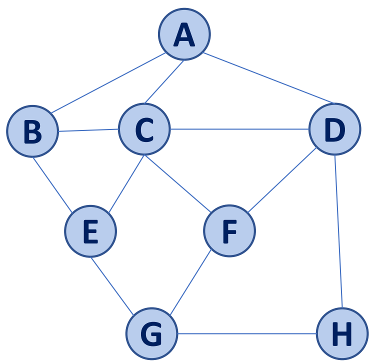


- Ordered
- Obvious Start
-



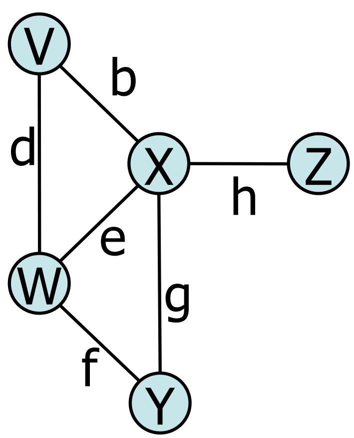
-
-
-

Traversal: BFS



Graph ADT

Data:
- **Vertices**
- **Edges**
- **Some data structure maintaining the structure between vertices and edges.**

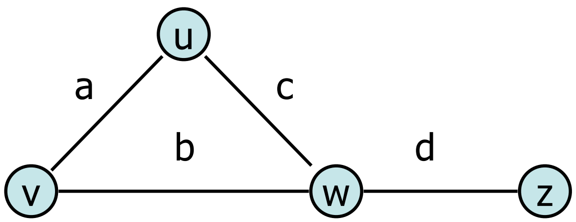


Functions:
- **insertVertex(K key);**
- **insertEdge(Vertex v1, Vertex v2, K key);**

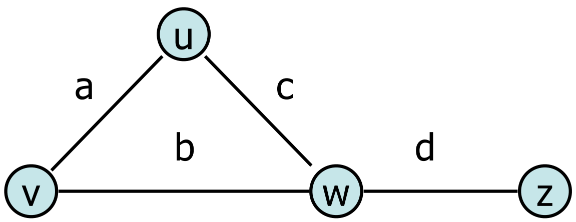
- **removeVertex(Vertex v);**
- **removeEdge(Vertex v1, Vertex v2);**

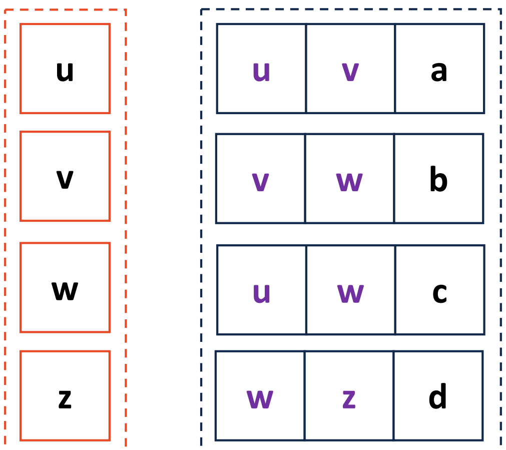
- **incidentEdges(Vertex v);**
- **areAdjacent(Vertex v1, Vertex v2);**

Graph Implementation Idea



Graph Implementation: Edge List

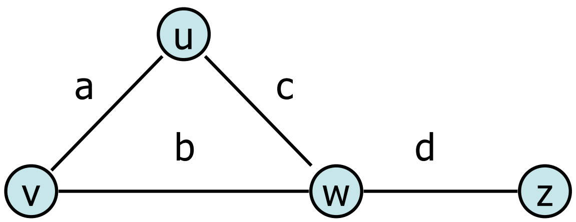




Vertex Collection:

Edge Collection:

Graph Implementation: Edge List



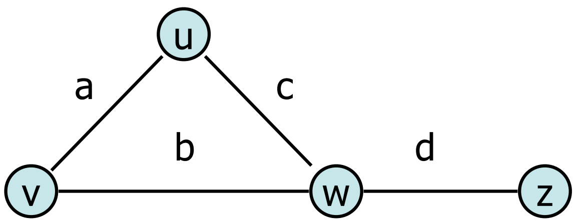
u
v
w
z

u	v	a
v	w	b
u	w	c
w	z	d

insertVertex(K key):

removeVertex(Vertex v):

Graph Implementation: Edge List



u
v
w
z

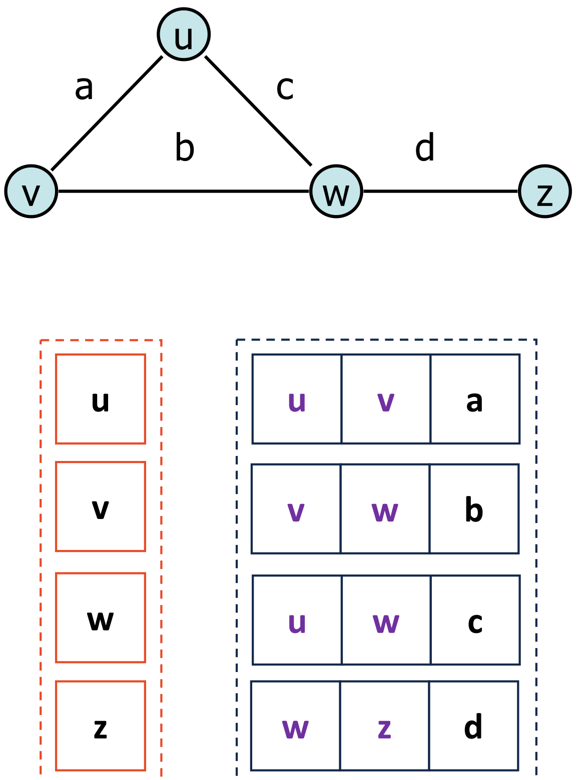
u	v	a
v	w	b
u	w	c
w	z	d

incidentEdges(Vertex v):

areAdjacent(Vertex v1, Vertex v2):

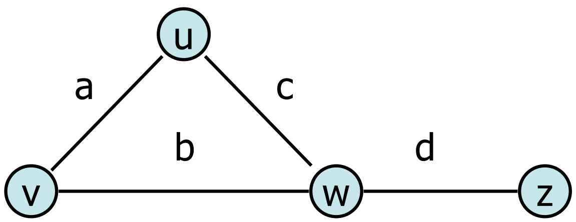
```
G.incidentEdges(v1).contains(v2)
```

Graph Implementation: Edge List



insertEdge(Vertex v1, Vertex v2, K key):

Graph Implementation: Adjacency Matrix

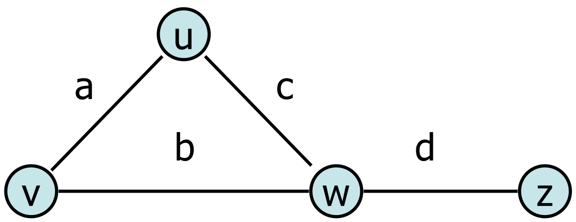


u
v
w
z

u	v	a
v	w	b
u	w	c
w	z	d

	u	v	w	z
u				
v				
w				
z				

Graph Implementation: Adjacency Matrix

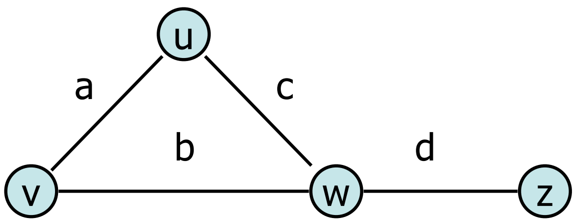


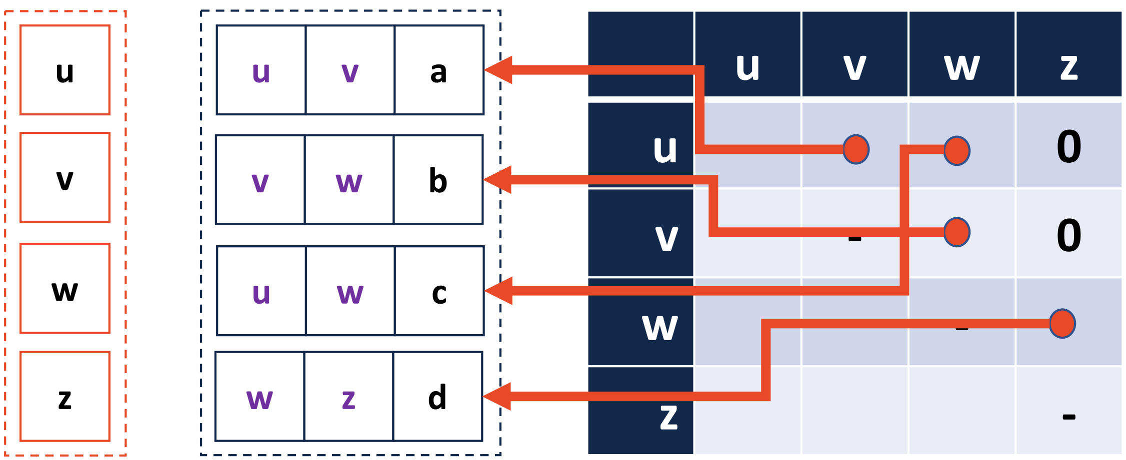
u
v
w
z

u	v	a
v	w	b
u	w	c
w	z	d

	u	v	w	z
u	-	1	1	0
v		-	1	0
w			-	1
z				-

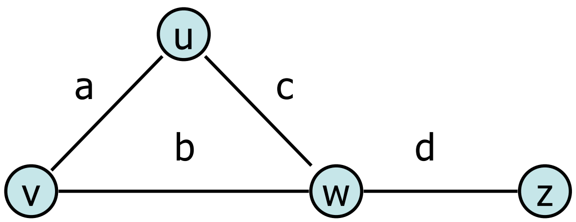
Graph Implementation: Adjacency Matrix

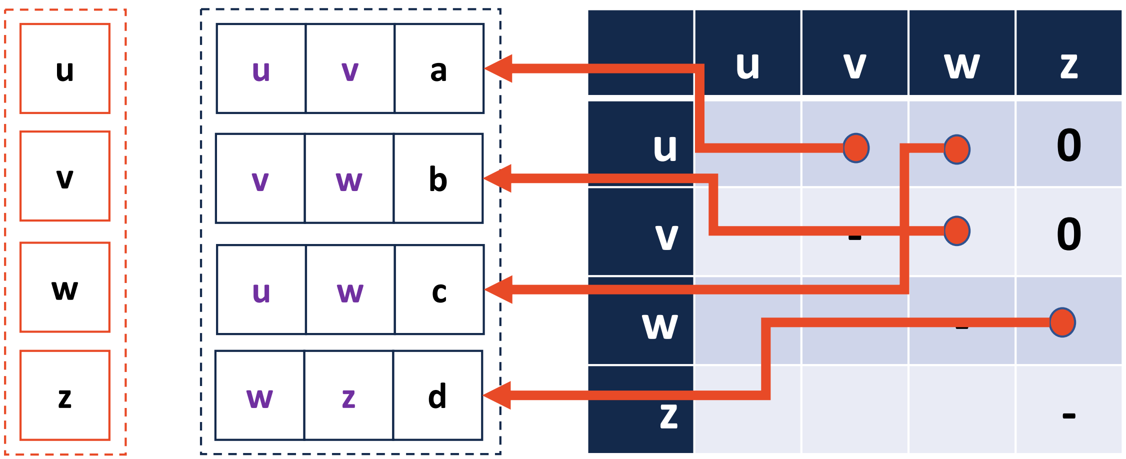




Graph Implementation: Adjacency Matrix

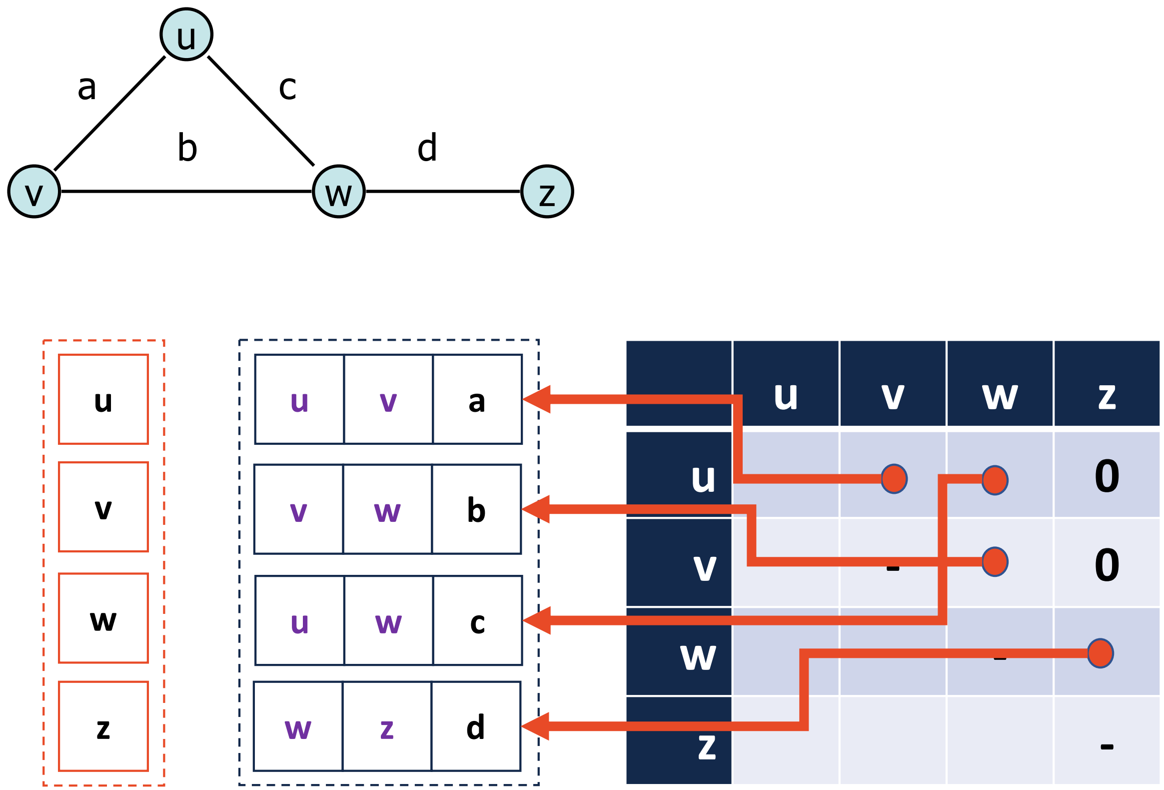
insertVertex(K key):





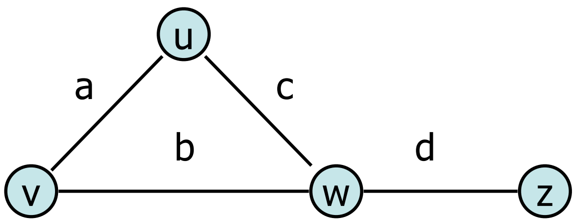
Graph Implementation: Adjacency Matrix

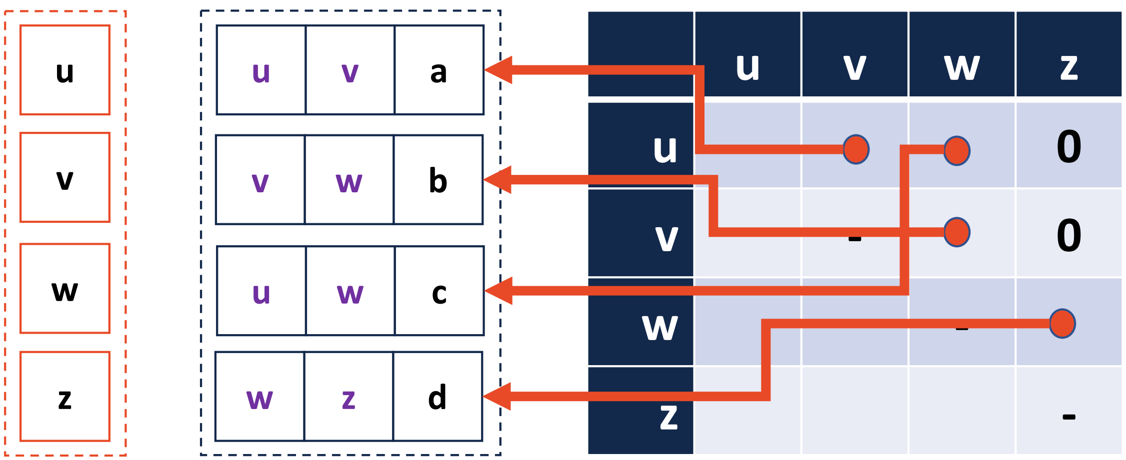
removeVertex(Vertex v):



Graph Implementation: Adjacency Matrix

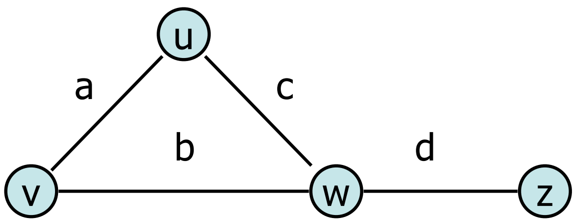
incidentEdges(Vertex v):

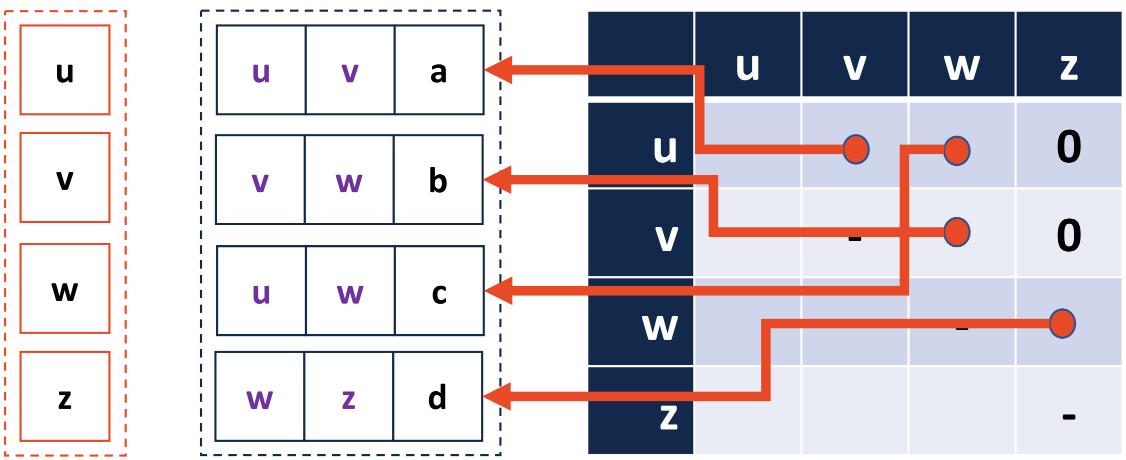




Graph Implementation: Adjacency Matrix

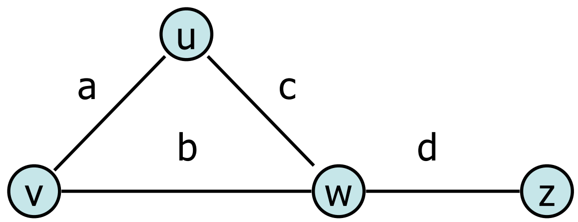
areAdjacent(Vertex v1, Vertex v2):

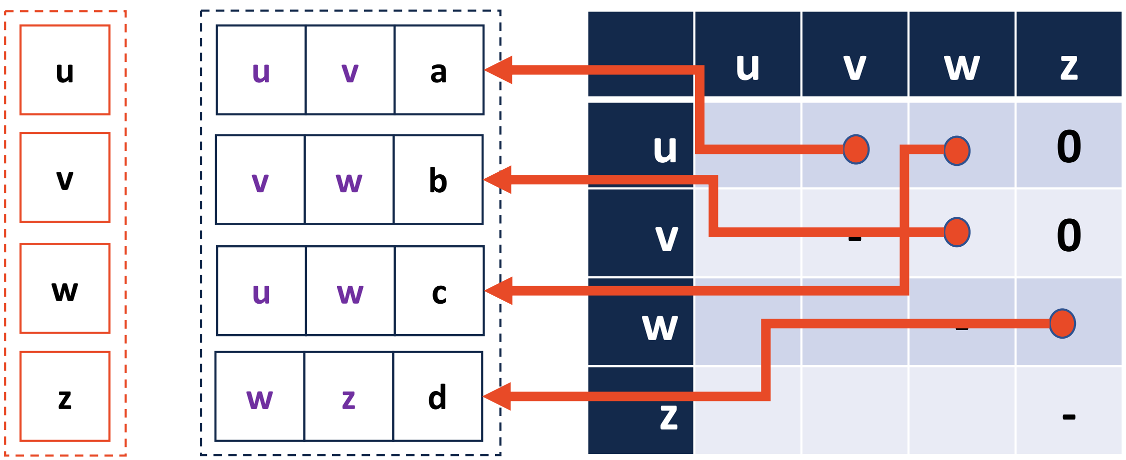




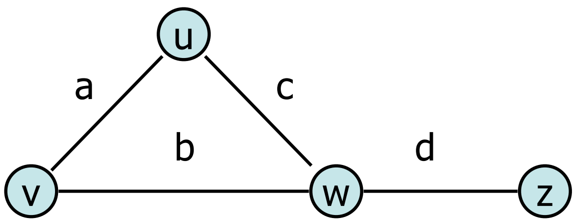
Graph Implementation: Adjacency Matrix

insertEdge(Vertex v1, Vertex v2, K key):





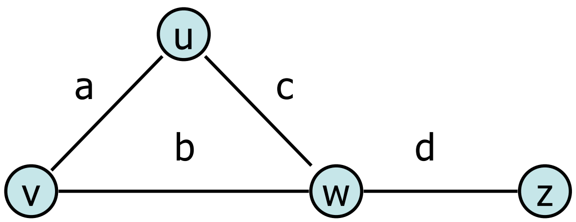
Graph Implementation: Edge List

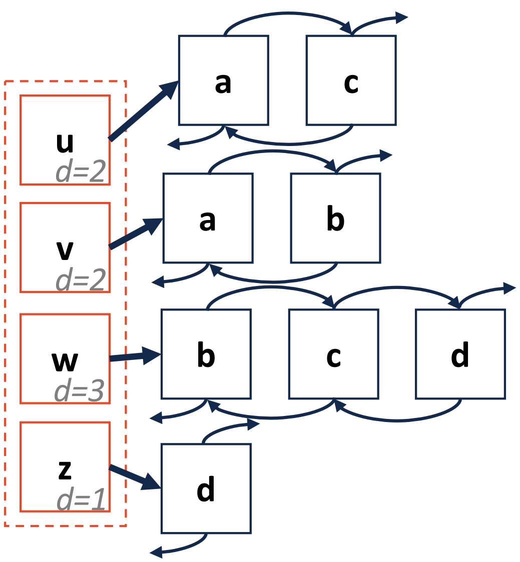


u
v
w
z

u	v	a
v	w	b
u	w	c
w	z	d

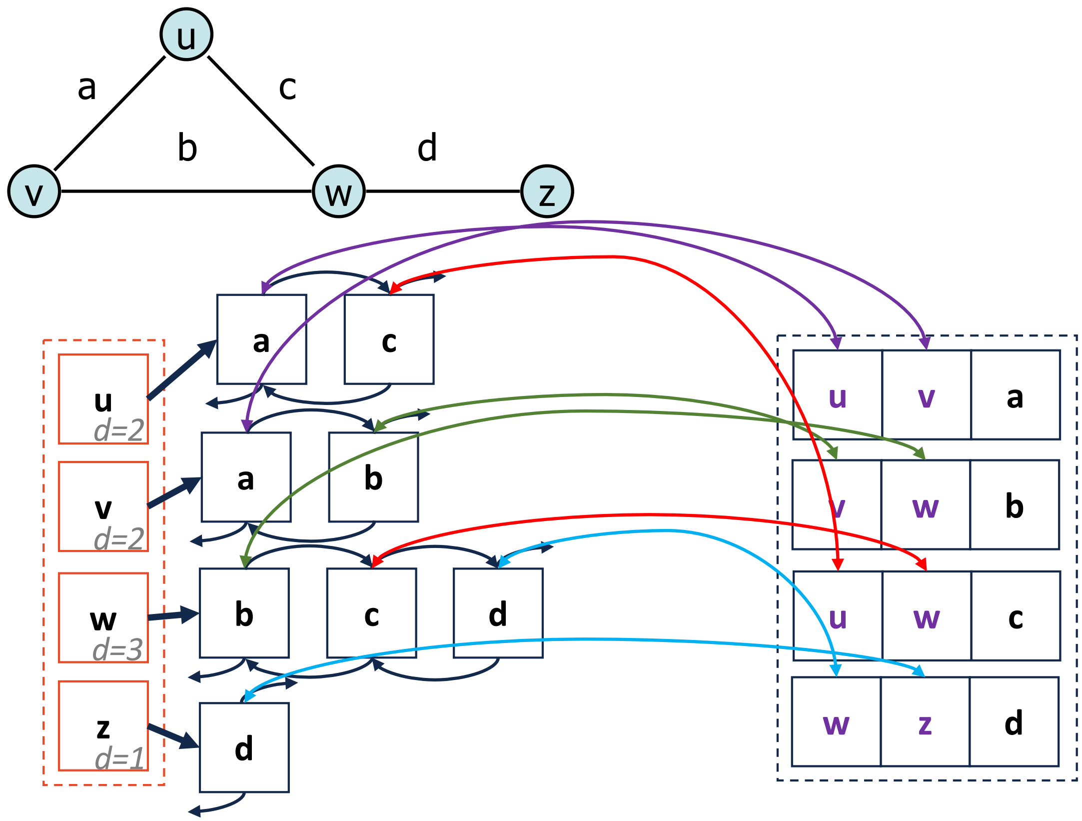
Adjacency List





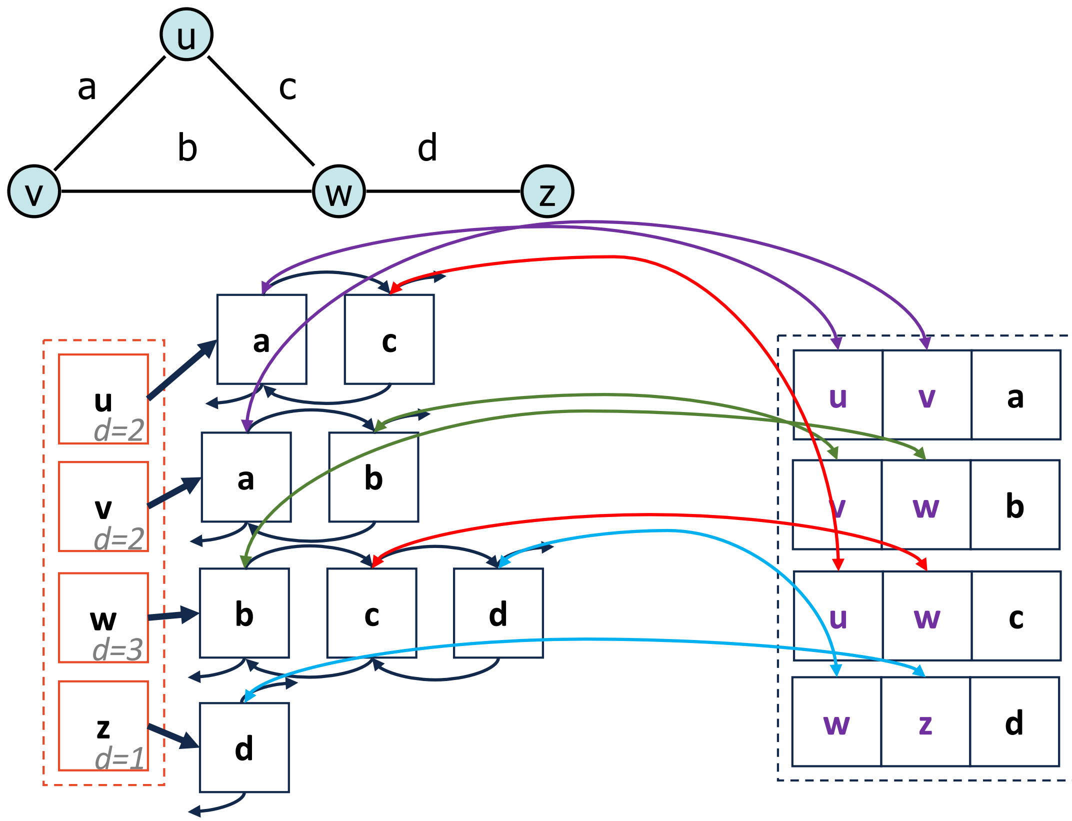
u	v	a
v	w	b
u	w	c
w	z	d

Adjacency List



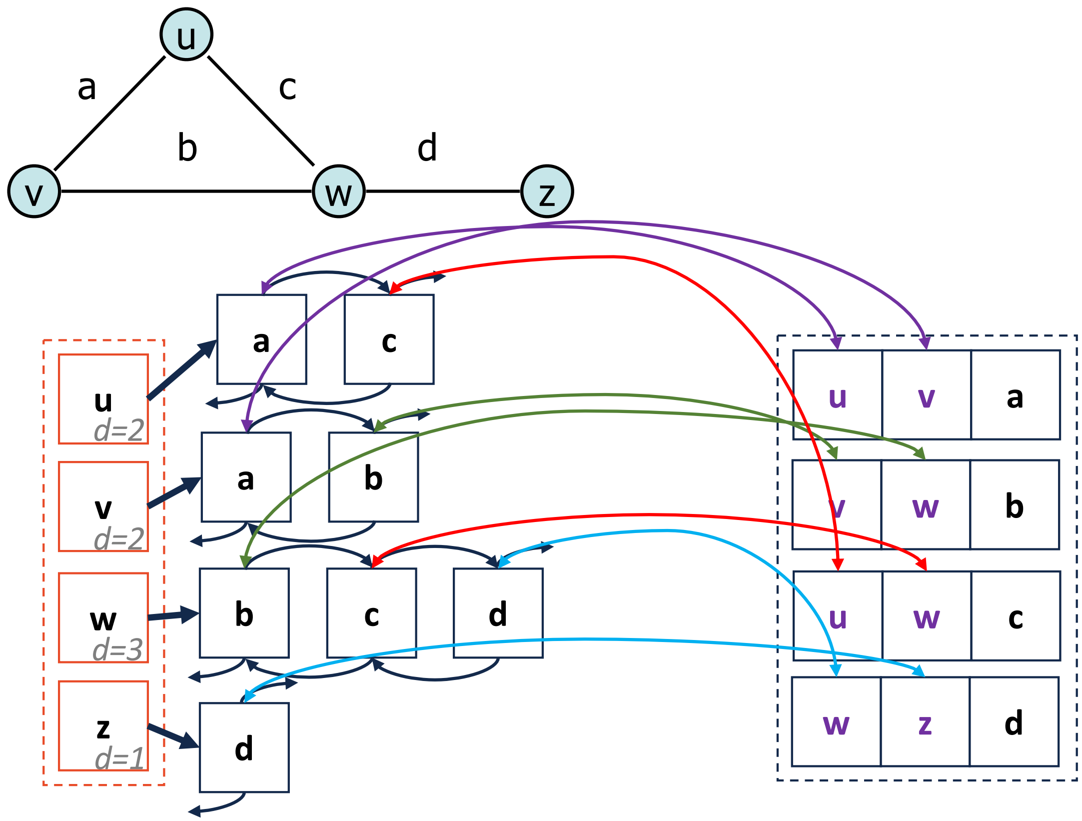
Adjacency List

insertVertex(K key):



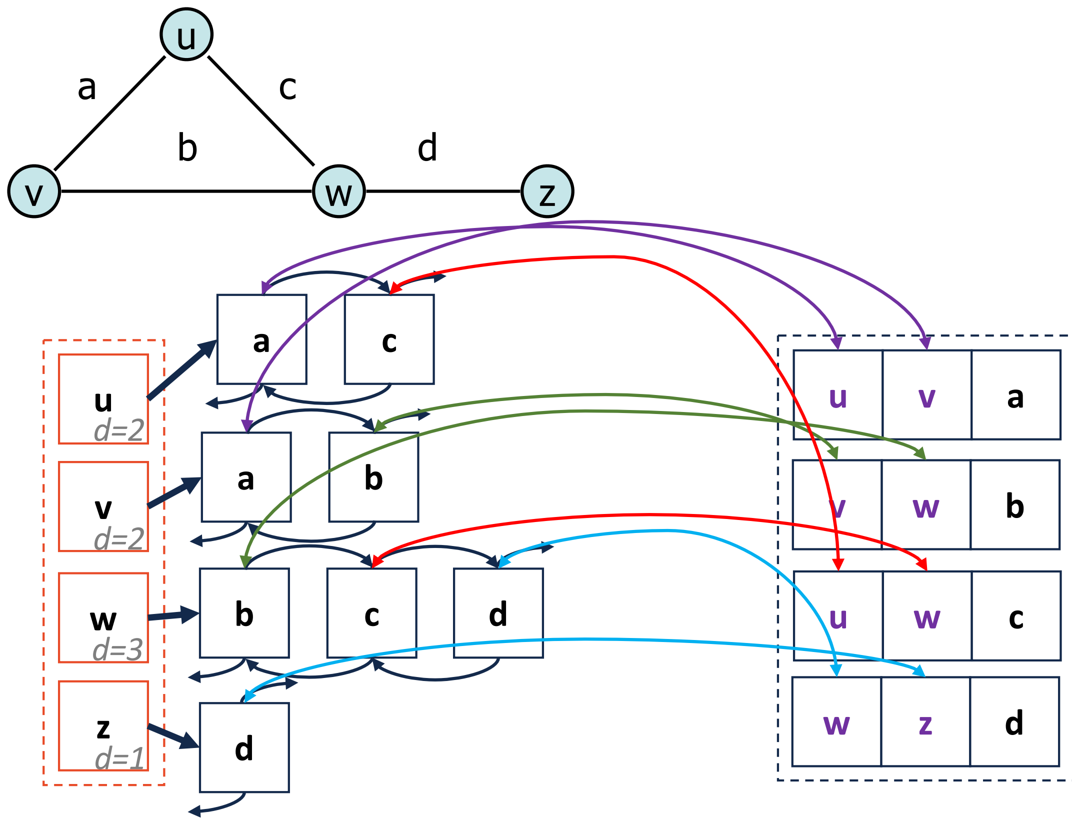
Adjacency List

removeVertex(Vertex v):



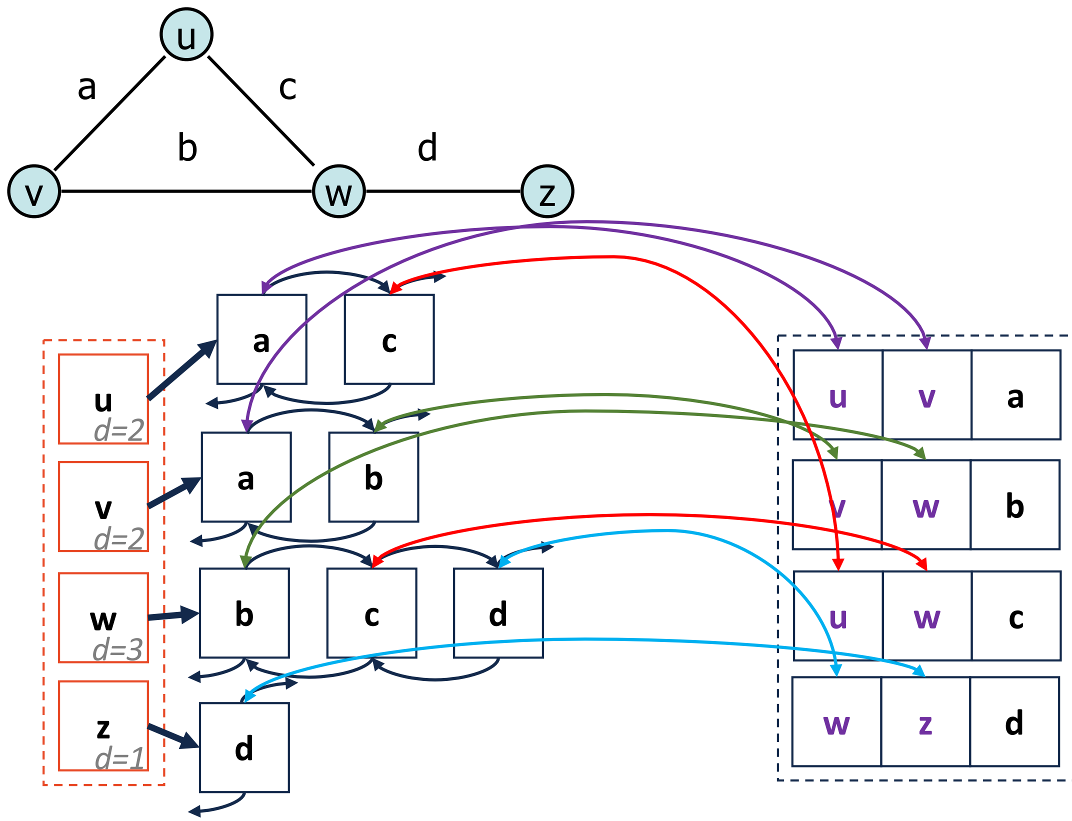
Adjacency List

incidentEdges(Vertex v):



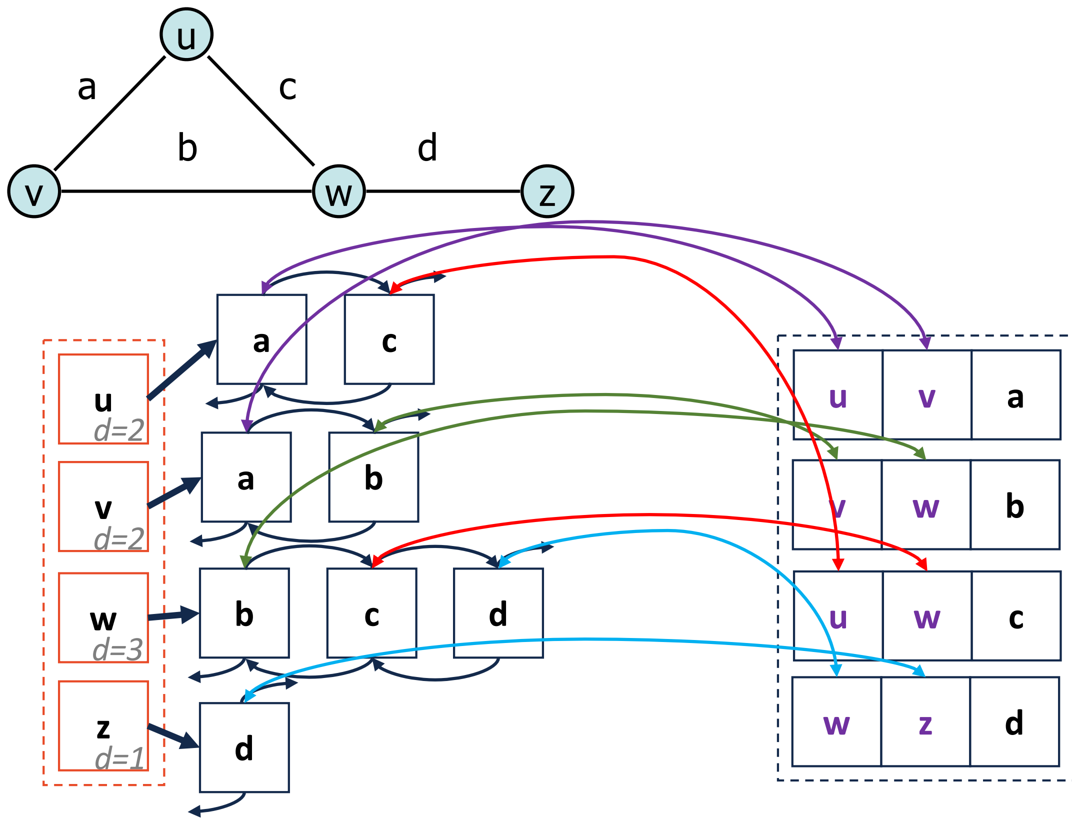
Adjacency List

areAdjacent(Vertex v1, Vertex v2):



Adjacency List

insertEdge(Vertex v1, Vertex v2, K key):



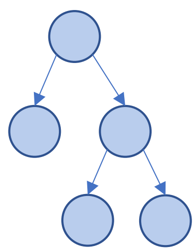
Expressed as O(f)	Edge List	Adjacency Matrix	Adjacency List
Space	n+m	n^2	n+m
insertVertex(v)	1	n	1
removeVertex(v)	m	n	deg(v)
insertEdge(v, w, k)	1	1	1
removeEdge(v, w)	1	1	1
incidentEdges(v)	m	n	deg(v)
areAdjacent(v, w)	m	1	min(deg(v), deg(w))

Traversal:

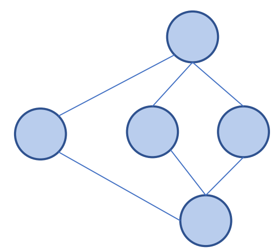
Objective: Visit every vertex and every edge in the graph.

Purpose: Search for interesting sub-structures in the graph.

We've seen traversal before ….but it's different:



- Ordered
- Obvious Start
-



-
-
-

Traversal: BFS